FOREWORD

This document will be reviewed and updated by the Defense Information Systems Agency (DISA) as required to remain current with technology and program requirements. This document supersedes all previous GCCS and DII Integration documents.

Changes to this document must be approved by DISA, but comments and recommendations for change may be forwarded for review and incorporation to:

DISA DII COE Chief Engineer
Attention: Mr. Dan Test
Department of Defense
Defense Information Systems Agency
45335 Vintage Park Plaza
Sterling, VA 20166-6701

Office Tel: (703) 735-8736
email: *testd@ncr.disa.mil*

Executive Summary

This document describes the technical requirements for using the Defense Information Infrastructure (DII) Common Operating Environment (COE) to build and integrate systems. It provides implementation details that describe, from a software development perspective, the following:

- the COE approach to software reuse,
- the COE runtime execution environment,
- the definition and requirements for achieving DII compliance,
- the process for automated software integration into the COE or into a COE-based system, and
- the process for electronically submitting/retrieving software components to/from the DII repository.

DII compliance is closely associated with interoperability, and for this reason systems are increasingly being measured by the degree to which they meet requirements described in this document. OSD has issued a directive that all new C4I systems and other systems which interface to C4I systems shall be in compliance with the *Joint Technical Architecture* (*JTA*). The *JTA* in turn mandates use of the DII COE. The *JTA* is being expanded in scope to address weapons systems as well.

Background

The DII COE concept is best described as an **architecture** that is fully compliant with the *DOD Technical Architecture for Information Management (TAFIM), Volume 3,* an **approach** for building interoperable systems, a **reference implementation** containing a collection of **reusable software** components, a **software infrastructure** for supporting mission-area applications, and a set of **guidelines, standards, and specifications**. The guidelines, standards, and specifications describe how to reuse existing software and how to properly build new software so that integration is seamless and, to a large extent, automated. The *JTA* replaces the standards guidance in the *TAFIM* as per OSD directive dated 22 Aug 96. In the absence of a Joint Systems Architecture, the *JTA* currently mandates the use of the DII COE (a fundamental JSA component) in Section 2.2. The DII COE will be evolved as necessary to maintain compliance with mandated standards found in future *JTA* updates.

The COE is primarily concerned with the executable environment of a system and is specifically designed to be programming-language neutral. It does not state a preference of one language over another, but leaves the selection of a programming language to higher-level standards profile guidance and programmatic considerations. Any statements in the *I&RTS* which appear to state or imply a preference for one language over another are unintentional.

The COE is a “plug and play” open architecture. The current reference implementation is designed around a client/server model. The COE is *not* a system; it is a *foundation* for building an open system. Functionality is easily added to or removed from the target system in small manageable units, called *segments.* Structuring the software into segments is a powerful concept that allows considerable flexibility in configuring the system to meet specific mission needs or to minimize hardware requirements for an operational site. Site personnel perform field updates by replacing affected segments through use of a simple, consistent, graphically oriented user interface.

The DII COE was initially based on work from the C4I arena, but it has been expanded to encompass a range of other functional areas including logistics, transportation, base support, personnel, health affairs, and finance. Three representative systems that use the DII COE are the Global Command and Control System (GCCS), the Global Combat Support System (GCSS), and the Electronic Commerce Processing Node (ECPN) system. All three systems use the same infrastructure and integration approach, and the same

COE components for functions that are common between the systems. GCCS is a C4I system with two main objectives: the replacement of the World-Wide Military Command and Control System (WWMCCS) and the implementation of the C4I For the Warrior concept. GCCS is already fielded at a number of operational CINCs and in calendar year 1996, achieved the first objective of replacing all WWMCCS systems. GCSS is under development and is targeted for the warfighting support functions (logistics, transportation, etc.) to provide a system that is fully interoperable with the warfighter C4I system. Implemented to its fullest potential, GCSS will provide both warfighter support to include reachback from deployed commanders into the CONUS sustaining base infrastructure, and cross-functional integration on a single platform. ECPN is also under development and is to provide the foundation for paperless exchange of business information, including funds transfer, using electronic media. A number of other programs that are in the early stages of development have committed to using the DII COE, and several programs have committed to migrating their existing systems to the DII COE.

The DII COE represents a departure from traditional development programs. It emphasizes incremental development and fielding to reduce the time required to put new functionality into the hands of the warrior, while not sacrificing quality nor incurring unreasonable program risk or cost. This development approach is sometimes described as a "build a little - test a little - field a lot" philosophy. It is a process of continually evolving a stable baseline to take advantage of new technologies as they mature and to introduce new capabilities. But the changes are done one step at a time so that the warfighters always have a stable baseline product while changes between successive releases are perceived as slight. This approach allows program managers the option of taking advantage of recently developed functions to rapidly introduce new capabilities to the field, or to synchronize with COE development at various checkpoints for those environments where incremental upgrades are not readily acceptable to the customer community.

DISA maintains the COE software and software from its own COE-based systems (e.g., GCCS, GCSS, ECPN) in an online configuration management repository called SDMS (Software Distribution Management System). This approach decreases the development cycle by allowing developers to receive software updates, or to submit new software segments, electronically. With appropriate security measures, installation costs are also reduced because operational platforms may be updated electronically across SIPRNET or other LAN networks.

New Features

This new release represents an upgrade to the previous version of this document, the DII COE *Integration and Runtime Specification* (*I&RTS*), version 2.0. It is intended to amplify and clarify sections that were previously unclear or incomplete, and to present a set of new capabilities. ***This new version is completely backwards compatible with the previous release of this document.*** There is ***no resultant reduction in the compliance of systems that have already been migrated under the previous version of this document***, although Appendix B has been reworked to make compliance checking easier.

It should also be noted that the *I&RTS* document contents and version number are entirely independent of the DII COE software release contents and version number. There is no direct correspondence between a particular version number of the *I&RTS* and the capabilities available in a version of the DII COE with the same version number. The *I&RTS* document describes the technical requirements for using the DII COE and therefore addresses the current and future capabilities of the DII COE. Portions of the *I&RTS* are always ahead of the DII COE software, addressing future capabilities and technological advances, so that developers can see where the DII COE is headed.

Several new capabilities are incorporated into this release including:

- Guidance for using DCE (Distributed Computing Environment)
- Extensions for World-Wide-Web (WWW) applications within the COE
- Database application support through the Shared Data Environment (SHADE)
- Inclusion of an NT-based COE for PCs
- Additional tools for managing large-scale LAN environments.

Conclusion

The principles described in this document are not unique to DISA programs. They can be readily applied to many application areas. The specific software components selected for inclusion in the COE determine the mission area that the COE can address. The concepts herein represent the culmination of open systems evolutionary development from both industry and government. Most notably, the Army Common Software (CS) and the Navy Joint Maritime Command Information System (JMCIS) COE efforts have greatly influenced DII COE development.

The DII COE architecture is an innovative framework for designing and building military systems. Because it reuses software contributed by mature programs, it utilizes field-proven software for common warrior functions. The engineering procedures for adding new capabilities and integrating systems are mature, and have been used for several Navy JMCIS releases as well as in all production GCCS releases. The end result is a strategy for fielding systems with increased interoperability, reduced development time, increased operational capability, minimized technical obsolescence, minimal training requirements, and minimized life-cycle costs.

This page is intentionally blank.

Introduction

The Command, Control, Communication, Computer, and Intelligence (C4I) For the Warrior (C4IFTW) vision has been stated as follows:

> *The Warrior needs a fused, real-time, true-picture of the battlespace and the ability to order, respond, and coordinate vertically and horizontally to the degree necessary to prosecute the mission in that battlespace.*

This broad visionary statement demonstrates that an unprecedented degree of integration and interoperability is required of Department of Defense (DOD) systems, both for legacy systems and for systems that are under construction. The Defense Information Infrastructure (DII) Common Operating Environment (COE) is the key to achieving this vision.

The DII COE[1] originated with a simple observation about command and control systems: certain functions (mapping, track management, communication interfaces, etc.) are so fundamental that they are required for virtually every command and control system. Yet these functions are built over and over again in

[1] The acronyms “DII COE” and “COE” are used interchangeably throughout this document. Other COEs exist (such as the Joint Maritime Information System (JMCIS) COE) which are very similar in scope or implementation with the DII COE. To avoid confusion, unless otherwise indicated, “COE” always refers to the DISA DII COE.

incompatible ways even when the requirements are the same, or vary only slightly, between systems. If these common functions could be extracted, implemented as a set of extensible low-level building blocks, and made readily available to system designers, development schedules could be accelerated and substantial savings could be achieved through software reuse. Moreover, interoperability would be significantly improved because common software is used across systems for common functions, and the functional capability only needs to be built correctly once rather than over and over again for each project.

This observation led to the development of the DII COE. Although its roots are in the C4I arena, the DII COE and its principles are not unique to C4I. The DII COE has been expanded to encompass a range of other functional areas including logistics, transportation, base support, personnel, health affairs, and finance. All new Defense Information Systems Agency (DISA) systems are being built using the DII COE while existing DISA systems are being migrated to use the DII COE. The Office of the Secretary of Defense (OSD) has recently issued a directive[2] that requires JTA compliance and, indirectly, use the DII COE.

A significant aspect of the COE challenge is to strategically position the architecture so as to be able to take advantage of technological advances. At the same time, the system must not sacrifice quality, stability, or functionality already in the hands of the warrior. In keeping with current DOD trends, the COE emphasizes use of commercial products and standards where applicable to leverage investments made by commercial industry.

[2] OSD Directive dated 22 August 1996 (Subject: Implementation of the DOD Joint Technical Architecture). The directive states that all new C4I systems and other systems which interface to C4I systems shall be in compliance with the *JTA*. The *JTA* in turn mandates use of the DII COE. The *JTA* is being expanded in scope to address weapons systems as well.

1.1 A Brief History of the DII COE

Initial DII COE development was driven by a near-term requirement to build a suitable WWMCCS replacement. To achieve the near-term WWMCCS replacement objective, technical experts and program managers from the Services, intelligence community, Defense Mapping Agency (DMA), and other interested agencies met for several months beginning in the fall of 1993. Participants proposed candidate systems as a possible starting point for a COE architecture or as a suitable candidate for providing capabilities to meet WWMCCS replacement requirements. None of the candidate systems met all requirements, but it was clear that a combination of the "best" from several systems could produce a near-term system that would be suitable for WWMCCS replacement. Moreover, an infrastructure could be put into place and a migration strategy defined to preserve legacy systems until migration to the intended architecture could be realized.

The cornerstone architectural concept jointly developed during that series of meetings was the GCCS COE. This initial COE was limited in scope to address the immediate C4I problem (i.e., WWMCCS replacement), but its principles, structure, and foundation deliberately went far beyond just the C4I mission domain. The GCCS COE was composed of software contributed from candidate systems evaluated by this original Joint engineering team.

An initial proof-of-concept system, GCCS 1.0, was installed in early 1994 at one site to validate the approach and to receive early feedback. GCCS 1.1 followed in the summer of 1994 and was the first attempt to integrate software from Service programs as initial GCCS COE components. GCCS 1.1 included mission applications from other programs operating in a "federated" mode. That is, the mission applications were integrated together so as to be able to run on the same hardware without interfering with each other, but not yet able to effectively share data between applications. GCCS 1.1 was installed and tested at beta sites and used at certain operational sites to monitor events during the 1994 Haiti crisis. GCCS 2.0 fielding began in early 1995 at a number of operational sites. GCCS 2.1 was fielded in mid-1995 and by mid-1996 had successfully replaced WWMCCS. A prototype version of GCCS 2.2 was the basis for Joint Warrior Interoperability Demonstration (JWID) 95 and a refinement of it was the basis for JWID 96. Another GCCS 2.2 enhancement was placed in theater to support Bosnia operations and for contingency planning when tensions in the Gulf area increased in mid-1996.

In mid-1995 technical experts met under DISA guidance to expand the GCCS COE into the DII COE. The result is a COE that contains all of the original GCCS COE functionality and that is backwards compatible. The DII COE was expanded to address other mission domains. Much of the original software has been updated to take advantage of further technological advances and Commercial Off-the-Shelf (COTS) software has replaced some of the original Government Off-the-Shelf (GOTS) components. From this historical perspective, the GCCS COE can be viewed as a subset of the much larger DII COE. Although GCCS succeeded in replacing the aged WWMCCS, it is important to realize that GCCS is far more than just a WWMCCS replacement.

The DII COE has its roots in command and control, but the principles and implementation described in this document are not unique to, nor limited to, command and control or logistics applications but are readily applicable to many other application areas. The specific software components selected for inclusion in the COE determine the mission areas that the COE can address.

Backwards compatibility is a fundamental tenet of the COE and significant effort is expended to preserve legacy investments. Systems which migrate to the DII COE now are protected by backwards compatibility as future COE versions are released. Upgrading from one COE version to the next is generally no more difficult than upgrading from one COTS product version to the next.

1.2 The DII COE Concept

The DII COE concept is a new approach that is much broader in scope than software reuse. Most software reuse approaches to date have proven less than satisfactory. Reuse approaches have generally emphasized the development of a large software repository from which designers may pick and choose modules or elect to rebuild modules from scratch. It is not sufficient to have a large repository, and too much freedom of choice leads to interoperability problems and duplication of effort. This rapidly negates the advantages of software reuse.

Software reuse strategies have also ignored the importance of data reuse. The approach has traditionally been to encapsulate data into a relational database from which applications may retrieve the data according to their own view (i.e., schema). While this approach was a tremendous advance, it fell short of the goal of providing truly interoperable systems in the Joint arena. What is required is an approach that promotes data sharing within systems and between systems. The approach must also recognize and resolve the issues of duplicative data, inconsistencies in the data, and data replication. SHADE is the data reuse strategy for the DII COE.

The DII COE emphasizes both software reuse and data reuse and interoperability for both data and software. But its principles are more far reaching and innovative. The COE concept encompasses:

- an architecture and approach for building interoperable systems,
- an environment for sharing data between applications and systems,
- an infrastructure for supporting mission-area applications,
- a rigorous definition of the runtime execution environment,
- a reference implementation on which systems can be built,
- a collection of reusable software components and data,
- a rigorous set of requirements for achieving DII[3] compliance,
- an automated toolset for enforcing COE principles and measuring DII compliance,
- an automated process for software integration,
- an approach and methodology for software and data reuse,
- a set of Application Program Interfaces (APIs) for accessing COE components, and
- an electronic process for submitting/retrieving software and data to/from the DII repository.

This document is an engineering specification that describes *how* modules must interact in the target system. System architects and software developers retain freedom in building the system, but runtime environmental conflicts and data conflicts are identified and resolved through automated tools that enforce COE principles. An important side effect is that traditional integration tasks largely become the responsibility of the developer. Developers are required to integrate and test their software with the COE prior to delivering it to the government. This simplifies integration because those who best understand the software design (the original developers) perform it, it reduces the cost because integration is performed earlier and at a lower level in the process, and it allows the government to concentrate on validation instead of integration.

The COE must be understood as a multi-faceted concept. Understanding how the many facets interact is important to appreciate the scope and power of the DII COE and to avoid confusion in understanding COE material. The next subsection deals with four specific facets in more detail:

- the COE as a system foundation,
- the COE as an architecture,

[3] The term “DII compliance” is preferred instead of “COE compliance” and is used throughout the *I&RTS*. The compliance concept and approach has not changed, but compliance is measured for segments within the COE as well as mission-application segments that lie outside the COE. Therefore, “DII compliance” is more descriptive and correct than “COE compliance.”

- the COE as a reference implementation, and
- the COE as an implementation strategy.

Failure to understand these facets will lead to confusion and non-compliant systems.

1.2.1 The DII COE as a System Foundation

The DII COE is *not* a system; it is a foundation for building systems. Figure 0-1 is a simplified diagram that shows how the DII COE serves as a foundation for building multiple systems. Details such as specific COE components, databases, and the internal structure of the COE have been omitted for clarity. Chapter 2 provides this level of information and describes the COE in much more detail. The purpose of Figure 0-1 is to introduce the concept.

The shaded box in Figure 0-1 shows two types of reusable software: the operating system and COE components. For the present discussion, it is sufficient to note that COE components are accessed through APIs and that the COE components form the architectural backbone of the target system. The API is the means through which a system permits a programmer to develop applications through interaction with the underlying COE. Standards (*POSIX* [*Portable Operating System for Information Exchange*] in the diagram) and specifications (*TAFIM* [*Technical Architecture Framework for Information Management*], *JTA* [*Joint Technical Architecture*], *I&RTS* [*Integration and Runtime Specification*], and *User Interface Specification* [*UIS*] in the diagram) dictate how COE components are to be built and how external components must be built to be compliant with the COE architecture.

Building a target system includes combining COE components with mission-specific software. The COE infrastructure manages the flow of data through the system, both internally and externally. Mission-specific software is mostly concerned with requesting data from the COE and then presenting it in a form that is most meaningful to the operator (e.g., as a pie chart, in tabular form, as a graph). The COE provides the necessary primitives for such data whether stored locally or remotely across a Local Area Network (LAN) or Wide Area Network (WAN). This frees the system designer to concentrate on meaningful data presentation and not on the mechanics of data manipulation, network communications, database storage, etc.

There is only one COE regardless of the target system. The COE is a set of building blocks. System designers select those building blocks (e.g., COE components) required for their mission application, while excluding building blocks that are not required. Each derived system uses the same set of APIs to access common COE components, the same approach to integration, and the same set of tools for enforcing COE principles. For common functions (e.g., communications interfaces, dataflow management), each target system uses precisely the same COE software components. Compliant systems do not implement their own versions of algorithms within the COE because this will rapidly lead to interoperability problems as algorithms are interpreted differently or because systems fail to upgrade algorithms at the same time. This approach to software reuse significantly reduces interoperability problems because if the same software is used, it is not possible to have two systems that interpret or implement standards differently.

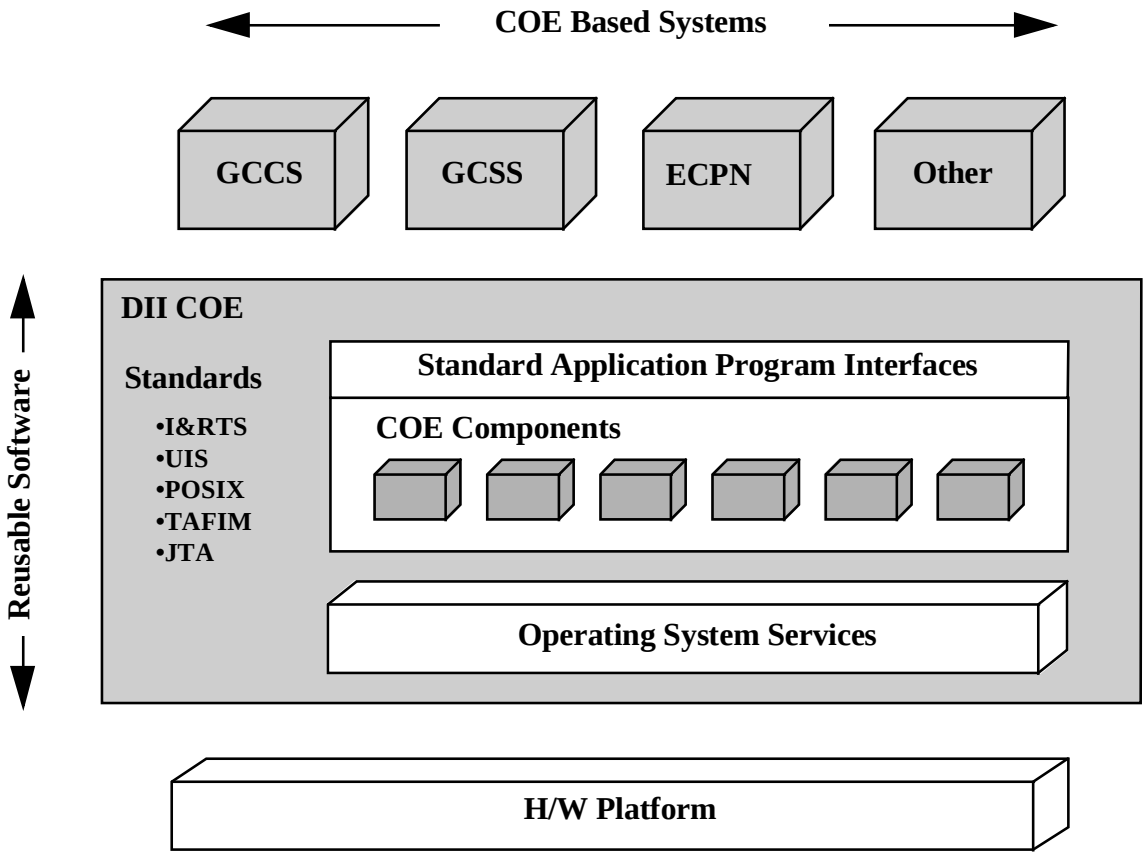


Figure 0-1: DII COE and COE-Based Systems

1.2.2 The DII COE as an Architecture

The DII COE is a "plug and play" open architecture designed around a client/server model. Functionality is easily added to or removed from the target system in small manageable units called *segments*. Segments are defined in terms of functions that are meaningful to operators, not in terms of internal software structure. Structuring the system into segments in this manner allows flexibility in configuring the system to meet specific mission needs or to minimize hardware requirements for an operational site. Site personnel perform field updates by replacing affected segments through use of a simple, consistent, graphically-oriented user interface.

The DII COE model is analogous to the Microsoft Windows® paradigm. The idea is to provide a standard environment, a set of standard off-the-shelf components, and a set of programming standards that describe how to add new functionality to the environment. The Windows paradigm is one of "federation of systems" in that properly designed applications can coexist and operate in the same environment. But simple coexistence is not enough. It must be possible for applications to share data. The DII COE extends the Windows paradigm to allow for true "integration of systems" in that mission applications share data at the server level.

Federation versus integration is an important architectural distinction. However, integration is not possible without strict standards that describe how to properly build components to add to the system. This applies equally to software functions and data. This document and other related documents detail the technical requirements for a well-behaved, DII-compliant application. The COE provides automated tools to measure compliance and to pinpoint problem areas. A useful side effect of the tools and procedures is that software integration is largely an automated process, thus significantly reducing development time while automatically detecting potential integration and runtime problem areas.

More precisely, to a developer the DII COE includes each of the following:

- **An Architecture[4]:** A precisely defined *TAFIM* and *JTA*-compliant, client/server architecture for how system components will interact and fit together and a definition of the system-level interface to COE components.

- **A Runtime Environment:** A standard runtime operating environment that includes "look and feel," operating system, and windowing environment standards. Since no single runtime environment is possible in practice, the COE architecture provides facilities for a developer to extend the environment in such a way as to not conflict with other developers.

- **A Data Environment:** A standard data environment that prescribes the rules whereby applications can share data with other applications.

- **A Reference Implementation:** A clearly defined set of already implemented, reusable functions. A set of reusable software and data is a cornerstone of the DII COE product.

- **A Set of APIs:** A collection of interfaces for accessing COE components. Thus, the COE is a set of building blocks in the same sense that X Windows and Motif are building blocks for creating an application's Graphical User Interface (GUI).

- **A Set of Standards and Specifications:** A set of rules that describe how to use the COE, how to construct segments, how to create a GUI, etc.

- **A Development Methodology:** A process for developing, integrating, and distributing the system and a process for sharing components with other developers. The COE emphasizes and encourages incremental development that has the advantage of quickly producing usable functionality.

1.2.3 The DII COE as a Reference Implementation

The COE necessarily includes an implementation of the components defined to be in the COE. The reference implementation is the key to reusability and interoperability. Use of the reference implementation provided is required to assure interoperability and is therefore a fundamental requirement for DII compliance. The reference implementation may change over time to take advantage of new technologies or to fix problem reports, but incremental improvements are introduced while preserving backwards compatibility.

The term *reference implementation* should be properly understood in the context of the DII COE. It means that a single body of code has been used as a starting point for implementing the COE on a specific

[4] The *JTA* describes three types of architectures: operational, technical, and system. The DII COE is relevant to all three types but does not and cannot provide a complete architectural definition for all three types. For example, the operational architecture also includes consideration of the command echelon and reporting structure. This is dictated by policy and is thus outside the scope of the COE. The DII COE is limited to addressing those aspects of an architecture that can be implemented in hardware and software as dictated by higher level standards, concept of operations, and service doctrine.

hardware platform and operating system. The only differences in the actual executable binary code are those that arise purely as a result of porting from one platform to another. The algorithms and the way the algorithms are implemented are identical from platform to platform.

1.2.4 The DII COE as an Implementation Strategy

The COE is also an evolutionary acquisition and implementation strategy. This represents a departure from traditional development programs. It emphasizes incremental development and fielding to reduce the time required to put new functionality into the hands of the warrior, while not sacrificing quality nor incurring unreasonable program risk or cost. This approach is sometimes described as a "build a little - test a little - field a lot" philosophy. It is a process of continually evolving a stable baseline to take advantage of new technologies as they mature and to introduce new capabilities. But the changes are done one step at a time so that the warfighters always have a stable baseline product while changes between successive releases are perceived as slight. Evolutionary development has become a practical necessity for many development programs because the traditional development cycle time is longer than the technical obsolescence cycle time. This approach allows program managers the option of taking advantage of recently developed functions to rapidly introduce new capabilities to the field, or of synchronizing with COE development at various points for those situations where incremental upgrades are not readily acceptable to the customer community.

The COE implementation strategy is carefully structured to protect functionality contained in legacy systems so that over time they can migrate to full COE utilization. Legacy systems must use only "public" APIs and migrate away from use of "private" APIs. Public APIs are those interfaces to the COE that will be supported for the life cycle of the COE. Private APIs are those interfaces that are supported for a short period of time to allow legacy systems to migrate from unsanctioned to sanctioned APIs. All new development is required to use only public APIs and use of any other APIs results in a non-DII compliant segment.

From the perspective of a system developer, whether developing a new application or migrating an existing one, the COE is an open client/server architecture that offers a collection of services and already-built modules for mission applications. Thus, the developer's task is to assemble and customize[5] existing components from the COE while developing only those unique components that are peculiar to particular mission's requirements. These additional mission-unique components must still adhere to the standards specified in the *JTA* and this document. In many if not most cases, this amounts to adding new "pull-down menu entries and icons."

[5] Customization is achieved in two ways: by omitting COE components that are not required and by configuring operational characteristics of the selected COE components. Customization does *not* mean the ability to change the functional operation of the component (a) outside the configurable items provided by the component or (b) outside the facilities provided by the component's APIs. When customizing the COE is discussed in this document, it must be understood in this context as a way of tailoring the COE to meet a specific mission need.

1.3 Lessons Learned

The COE as the embodiment of an architectural concept offers the opportunity to leverage a mature, proven, field tested software base for a wide variety of applications for the services, agencies, and Joint community. As budgets shrink and as budgetary priorities shift, program managers require the ability to continue to respond rapidly with systems that satisfy the information needs of United States and Allied Armed Forces. The COE implementation strategy is a significant advancement in fulfilling this ongoing need.

Examination of state-of-the-art development in light of these realities results in a set of fundamental tenets that greatly influence the history, future, and direction of the DII COE. An explanation of these tenets is useful in understanding the COE as a whole.

- *Pre-COE practices lead to development and redevelopment of the same functionality across systems*. Redevelopment is frequently necessary because of technological changes as algorithms are improved or as hardware becomes faster and cheaper. However, development cost tends to be high due to a lack of coordination between programs that share common requirements.

- *Duplication of functionality within the same system is more expensive than avoiding duplication*. Lack of coordination between program developers is a fundamental cause for duplicative functions, but an additional factor is that reuse libraries are not commonly available. The impact of duplication is more than just program costs. When functionality is duplicated, system users are often given conflicting information even in the presence of identical data because designers took slightly different approaches to solving the same problems or made slightly different assumptions.

- *Interoperability is not achievable through "paper" standards alone*.[6] Standards are necessary, but not sufficient,[7] to guarantee interoperability. Interoperability problems are generally not caused by the standards chosen but by differing or incorrect interpretations of standards. System designers often choose different standards with which to comply, but even when the standards are the same, different interpretations of the standards can greatly change the way the resulting system operates. The COE emphasizes use of industry and government standards, but relies even more on automated ways of measuring and evaluating compliance, and thus quantitatively evaluating program risk. The only practical way to achieve interoperability is to use exactly the same software, written to appropriate standards, for common functions across applications. For example, the COE contains a common tactical track correlator to ensure that all users see the same tactical picture. The answer produced by the correlator may be incorrect but a problem correction in one place then becomes effective for all users.

- *Pre-COE practices lead to exponential growth in testing and associated development costs*. Lack of commonality and modularity in system building blocks means that there is much duplication of effort in testing basic functionality and testing in one section of a system is often tightly coupled to testing in another section. This complicates and extends the certification process. Configuration management, system integration, and long-term maintenance are also more complex and costly when there is a lack of commonality and modularity in system building blocks.

- *The importance of training is usually underestimated and the magnitude of the training problem is increasing.* An operator is often expected to use multiple systems which behave completely differently,

[6] This statement is not meant to minimize the importance of standards, but to state that they alone are not sufficient to solve interoperability problems. The situation would be far more desperate in the absence of standards.

[7] The solution provided by the COE is to define specifications and a reference implementation of a standard. For example, in the user interface area, Motif is the standard selected for UNIX platforms and the *DII User Interface Specification* is the specification written to be compliant with Motif, but tailored for the particular mission domain.

are equally complex with their own subtleties, and which give slightly different answers. Operator turnover is rapidly reaching the point where the time it takes to train an operator is a significant portion of the time that the operator is assigned to his current tour of duty. Training is greatly reduced by a consistent "look and feel" and by the ability to present to the operator only those functions useful for the task at hand.

- *Don't reinvent the wheel*. If a component already exists, it should probably be utilized even if the component is not the optimum solution. Almost any module can be improved but that is rarely the issue. Reuse of existing and proven software allows focus of attention on mission uniqueness. Rather than concentrating scarce development resources on recreating building blocks, the resources can be more appropriately applied to configuration and development of functionality that is not already available.

- *Utilize existing commercial standards, specifications, and products whenever feasible*. The commercial marketplace generally moves at a faster pace than the military marketplace and advancements are generally available at a more rapid rate. Use of commercial products has several advantages. Using already built items lowers production costs. The probability of product enhancements is increased because the marketplace is larger. The probability of standardization is increased because a larger customer base drives it.

1.4 Requirements and Objectives

The following requirements apply to the DII COE:

- The DII COE will be fully compliant with the *JTA*[8]. Standards defined within the *JTA* promote an open systems architecture, the benefits of which are assumed to be well known and generally accepted.

- The DII COE is intended to be hardware independent and operate on a range of open systems platforms running under standards-based operating systems. Program-driven requirements, associated testing costs, and funding will dictate which specific hardware platforms are given priority.

- Non-developmental items (NDIs), including both COTS and GOTS products, are the preferred implementation approach.

- The DII COE is programming-language neutral. It does not state a preference of one language over another, but leaves the selection of a programming language to higher-level standards profile guidance and programmatic considerations. Any statements in the *I&RTS* which appear to state or imply a preference for one language over another are unintentional.

COE development is driven by C4IFTW requirements as articulated by the services through the appropriate DISA Configuration Control Board (CCB) process. Development priorities are established by the CCB Chair and given to the DII COE Chief Engineer for implementation.

The broad program drivers for the DII COE lead to a number of program objectives that include those stated in the *TAFIM, Volume 2*:

1. **Commonality**: Develop a common core of software that will form the foundation for Joint systems, initially for C4I and logistics systems.

2. **Reusability**: Develop a common core of software that is highly reusable to leverage the investment already made in software development across the services and agencies.

3. **Standardization**: Reduce program development costs through adherence to industry standards. This includes use of commercially available software components whenever possible.

4. **Engineering Base**: Through standardization and an open architecture, establish a large base of trained software/systems engineers.

5. **Training**: Reduce operator training costs and improve operator productivity through enforcement of a uniform human-machine interface, commonality of training documentation, and a consistent "look and feel."

6. **Interoperability**: Increase interoperability through common software and consistent system operation.

7. **Scalability**: Through use of the segment concept and the COE architectural infrastructure, improve system scalability so that COE-based systems will operate with the minimum resources required.

8. **Portability**: Increase portability through use of open systems concepts and standards. This also promotes vendor independence for both hardware and software.

[8] *JTA* replaces some of the standards guidance in the *TAFIM* as per OSD directive (Subject: Implementation of the DOD Joint Technical Architecture) dated 22 August 1996. It replaces those standards for service areas defined within the *JTA*. For those service areas not included in the *JTA*, guidance in Volume 7 of the *TAFIM* is to be followed.

9. **Security**: Improve system security to the extent possible to protect the system from deliberate attack and prevent unauthorized access to data and applications.

10. **Testing**: Reduce testing costs because common software can be tested and validated once and then applied to many applications.

1.5 Document Scope

This document describes the technical requirements for building and integrating software components on top of the DII COE. It provides implementation details that describe, from a software development perspective, the following:

- the Common Operating Environment (COE) approach to software reuse,
- the runtime execution environment,
- the Shared Data Environment (SHADE),
- the requirements for DII compliance,
- how to structure components to automate software integration, and
- how to electronically submit/retrieve software components to/from the software repository.

1.6 Applicable Documents, Standards, and Specifications

This document is one in a series of related documents that define development requirements, system architecture, engineering tools, and implementation techniques. Many of the documents cited are available on the World-Wide-Web (WWW), or contact the DISA Configuration Management (CM) office for information on how to obtain the desired documents.

Because the COE and COE-based systems are ongoing programs, enhancements and additional features are developed on a regular basis. Documentation updates are regularly released for each of the documents listed here. Be sure to always refer to the latest version for the documents listed below, and be aware that many of the documents are being modified and extended to address DII COE-based systems, not just GCCS or GCSS.

1. *Architectural Design Document for the Defense Information Infrastructure (DII) Common Operating Environment (COE)***, January 1996, DISA Center for Computer Systems Engineering**. This document is the definitive high-level technical description of the COE. It documents the architectural design produced by the DISA COE Design Working Group. It is useful for understanding how the client/server model has been implemented within the DII COE.

2. *C4ISR Architecture Framework***, CISA-0000-104-96, Version 1.0, 7 June 1996, C4ISR Integration Task Force (ITF) Integrated Architectures Panel**. This document presents an innovative definition of levels of interoperability. The DII COE adopts these levels of interoperability and maps DII compliance to interoperability levels.

3. *Defense Information Infrastructure (DII) Common Operating Environment (COE) Version 3.0 Baseline Specifications***, 31 October 1996, DISA**. This document describes the detailed contents of each COE release and is updated with each subsequent release. It includes the name and version of each segment in the COE as well as COTS products, their version, and applicable patches.

4. *Defense Information Infrastructure (DII) Common Operating Environment (COE) System Requirements Specification***, Draft, 1996, Institute for Defense Analysis**. Service and Agency requirements for a COE are defined in this document. It is a living document that is updated as necessary to reflect ongoing requirements collection.

5. *Defense Information Infrastructure Software Quality Compliance Plan,* **Draft, 1 January 1996, DISA**. This document describes a plan for evaluating COE segments from a software quality perspective. The plan includes static analysis of segment source code to measure complexity, maintainability, risk, and other standard software metrics.

6. *Department of Defense Joint Technical Architecture, Final Coordination Draft 1.0***, 22 August 1996, Joint Technical Architecture Working Group**. The *JTA* has been mandated by OSD directive for "... all emerging systems and systems upgrades. The *JTA* applies to all C4I systems and the interfaces of other key assets (e.g., weapons systems, sensors, office automation systems, etc.) with C4I systems. The *JTA* also applies to C4I Advanced Concept Technology Demonstrations and other activities that lead directly to the fielding of operational C4I capabilities." The *JTA* stipulates DII compliance as part of its requirements. It also "... replaces the standards guidance in the *Technical Architecture Framework for Information (TAFIM)* currently cited in DOD Regulation 5000.2-R."

7. *Department of Defense Technical Architecture Framework for Information Management, Volumes 1-8***, Version 3.0, 2 January 1997, DISA Center for Architecture**. This multi-volume document defines a standards profile and the DOD *Technical Reference Manual (TRM)* for information management systems. This document set also presents a high-level technical architecture that is useful for classifying levels within a system's infrastructure. The *TRM* distinguishes between the hardware platform, hardware-specific services, supporting infrastructure services, and mission applications.

8. *Information Technology - Portable Operating System Interface for Computer Environments (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, ISO 9945-1, 1990; *Information Technology - Portable Operating System Interface for Computer Environments (POSIX) - Part 2: Shell and Utilities***, ISO 9945-2, 1993**. The POSIX documents are an ongoing standardization effort that is attempting to define a common set of low-level functions, especially at the operating system level, across all hardware platforms and operating systems.

9. *User Interface Specification for the Defense Information Infrastructure (DII)***, Version 2.0, 1 April 1996, DISA**. This document, sometimes called the *DII Style Guide*, defines the "look and feel" of the user interface for COE-based systems. The *User Interface Specification* provides specifications for applications using Motif and Windows GUIs; a future version of the document will include Windows NT and Web-based applications.

1.7 Document Structure

This document is structured to correspond to the typical phases in a development cycle, beginning with how a developer builds a segment, submits it to the government, and then how it is fielded to an operational site. Chapter 1 of this document is an overview of the DII COE, a brief history of its development, and applicable documents and standards.

Chapter 2 gives a brief technical description of the COE, its components, and the principles that determine whether a software component is part of the COE or is a mission application. Selection of the particular components to populate the COE determine what applications can be supported, but the principles which define a COE are not application-specific. Chapter 2 also describes the important concept of DII compliance and maps compliance to levels of interoperability.

Chapter 3 is an overview of the development process. It includes a discussion of the process from segment registration through development, submission to DISA, integration, and site installation. The tools provided in the COE and how they are used is key to understanding automated integration.

Chapter 4 describes SHADE and other database considerations within the context of the COE. Databases are heavily used within COE-based systems, and early consideration of their structure, how they are to be used, and how they are to fit into the overall system is crucial in building a successful system.

Chapter 5 describes the runtime environment as it exists for operational sites, the disk directory and file structure fundamental to the COE, and the procedures for integrating segments into a runtime environment. Requirements detailed in Chapter 5 must be carefully followed so that applications will not interfere with each other, and so that integration is largely an automated process.

Chapters 6, 7, and 8 are new with this version of the *I&RTS*. They describe extensions for the COE reference implementation that runs on NT platforms, extensions to the COE to support Web applications, and support for Distributed Computing Environment (DCE) applications respectively.

Chapter 9 provides some suggestions for setting up a software development environment. Few requirements are stipulated for a development environment, allowing as much freedom for developers and program managers as possible.

Chapter 10 describes two important components for both developers and operational sites: the online COE Software Distribution Management System (SDMS), and the COE Information Server (CINFO). These components are used to disseminate and manage software, documentation, meeting notices, and general information of importance to the COE community.

Appendix A lists the currently supported COE configurations. The appendix includes supported hardware, and supported COTS versions. It also describes the Reference Implementation program whereby vendors may obtain low-level components of the COE and port them to their hardware platforms.

Appendix B presents a checklist for developers to use as an aid in determining the degree to which a segment is DII-compliant. As described in the appendix, some conditions are mandatory, others require a migration strategy to show conformance, while others are optional but recommended. This appendix has been reworded and reformatted to be clearer and easier to apply, but is otherwise unchanged from the previous *I&RTS* version.

Appendix C describes the automated tools provided with the COE. A number of new tools are provided to simplify the segment development and maintenance life cycle. The philosophy is to provide developers with access to the same tools that integrators will use so that segment integration is performed, as much as possible, by segment developers prior to segment delivery. Integration of segments with the COE is the responsibility of the segment developer. Government integrators serve as validators *only* in this process to ensure that developers produce DII-compliant segments. In addition to segment validation, government

integrators perform system-level integration of all segments submitted by all developers to create the target system.

Appendix D gives additional information on the COE online repository (SDMS) and the COE information server (CINFO).

Appendix E describes how to register a segment and what information is required for registration. Segment registration is required in order to identify potential conflicts as early in the development cycle as possible.

The remaining appendices provide additional information on products within the COE, such as the Relational Database Management System (RDBMS), that are either vendor-specific or product-version-specific.

Finally, a List of Acronyms used in the *I&RTS* are presented and a Glossary of frequently encountered terms. The acronyms and terms are encountered throughout DII COE-related documents.

This page is intentionally blank.

2. The DII COE

The concept of a COE as embodied in the DII COE is perhaps the most significant and useful technical byproduct of the Joint Service/Agency technical meetings that led to the successful GCCS development effort. It represents the culmination of several years of development amongst the services/agencies and it is interesting to note that the services/agencies independently arrived at similar conclusions. The DII COE encompasses architecture, standards, specifications, software reuse, shareable data, interoperability, and automated integration in a cohesive framework for systems development. Automated integration is described more fully in Chapter 3.

This chapter is devoted to explaining the DII COE in detail. Definition of a COE is principles-driven, not application-driven, so this chapter begins with a discussion of those principles. Selection of the actual components to populate the COE creates a COE *reference implementation*[9]. This is important because the components which constitute a COE instantiation determine the specific mission domain that a COE can address (e.g., C4I for GCCS, logistics for GCSS, finance for ECPN), and how broadly defined the mission domain can be. Because the COE is structured so that only required components are loaded, a properly defined COE is suitable for a service-specific system (e.g., Navy JMCIS, Air Force Battlefield Situation Display [BSD]) or a joint system (e.g., GCSS, ECPN). Also, because the architecture is principles-driven, the DII COE is extensible to larger mission domains by expanding the selected set of software components. The COE is an open architecture whose principles apply equally well to UNIX[10] and non-UNIX platforms such as the Personal Computer (PC). The DII COE contains a reference implementation for both UNIX and NT platforms.

Subsection 2.1 discusses fundamental COE concepts and also describes what is meant by DII compliance and interoperability. As with any standard, compliance is required to avoid conflicts that prevent interoperability. Take careful note in reading subsection 2.1 that the discussion is relevant to any COE-based system since the principles apply much more broadly than to a single system such as GCCS or GCSS. Remaining subsections elaborate on software and hardware configurations selected for support by

[9] Reference implementation means that an implementation of the COE exists and has been used as the basis for producing the same functional equivalent on other platforms. It does *not* imply that developers will be provided with source code to the COE and thus be responsible for porting it to other platforms.

[10] UNIX in this document is used in the sense of a vendor-proprietary implementation of the "traditional" UNIX operating system. Although desirable, it is not necessary that vendors have received an X/Open UNIX 95 branding.

DISA and how the software is structured at a top level to limit site operators to only those functions they are authorized to access.

2.1 Fundamental COE Concepts

In COE-based systems, all software and data - except certain portions of the kernel (see subsection 2.1.2.1) such as the operating system and basic windowing software - are packaged in self-contained units called *segments*. This is true for COE infrastructure software and for mission-application software as well. Segments are the most basic building blocks from which a COE-based system can be built. Segments are defined in terms of the functionality they provide, not in terms of "modules," and may in fact consist of one or more "modules." They are defined as a collection of related functions as seen from the perspective of the end user, not the developer. The reason for defining segments in this way is that it is a more natural way of expressing and communicating what software features are to be included in, or excluded from, the system than by individual process, file name, or data table. For example, it is more natural to think of a system as containing a message processing segment than executables called `MP_In` and `MP_Out`. It is more natural to the end user to think of a word processor segment than a software module that opens a file, another module that paginates a file, another module that compresses a file, etc.

Those segments that are part of the COE are known as *COE-component segments*, or more precisely, as segments that further have the attribute of being contained within the COE. Segments that are built on top of the COE to provide capabilities specific to a particular mission domain are *mission-application segments*. The principles which govern how segments are loaded, removed, or interact with one another are the same for all segments, but COE-component segments are treated more strictly because they are the foundation on which the entire system rests. A later chapter further refines the segment concept to distinguish between data segments, software segments, patches, etc. but the point here is that segments are a technique for packaging system components.

Each segment in the system contains a directory with a collection of data files that "self-describe" the segment to the rest of the COE. The directory that contains these files is called the *segment descriptor directory* and the files themselves are called *segment descriptors*. The process of decomposing a component into individual packages and creating the required segment descriptors is called *segmentation*.

Packaging a system in terms of segments along with the strict rules which govern the COE and runtime environment provide several immediate benefits:

- *Segment developers are decoupled and isolated from one another.* Segments are self-contained within an assigned directory. Developers have maximum freedom within the assigned segment directory, but minimum freedom outside it. This allows multiple developers to work in parallel with support for seamless integration after development.

- *Extensions to the environment provided by the COE are coordinated through automated software tools.* It is not possible to create a single configuration of the COE that meets all possible mission-application or site-unique requirements. However, the COE tools make it possible to extend the environment provided by the COE in a carefully controlled way to ensure compatibility and identify segment dependencies and conflicts.

- *Compliance verification and installation can be automated.* Standards without automated validation are difficult to use in practice, especially in a program where the system is large and there is a need to coordinate activities from several different contractors, program sponsors, services, and agencies. The COE approach to validation is closely related to software installation so that automation of one directly leads to automation techniques for the other.

- *Mission-application segments are isolated from the COE.* System integration problems are frequently a result of an undisciplined interaction between software components or because of tight coupling between components. The COE controls interaction through APIs and isolates mission applications from the COE-component segments so that failure of one mission-application segment is less likely to affect another or affect the stability of the COE foundation itself.

- *Segments created by one developer for one system can be readily reused by another developer for another system.* That is, the DII COE is an effective strategy that includes not just software reuse, but also ensures that a reused segment fits seamlessly into the new system.

- *Integration is simplified and the original developers resolve most integration problems before the segment is ever submitted.* The segment descriptors "self-describe" the segment so that all pertinent information required to integrate the segment into a system is contained in a standard, known location. The tools that validate conformance to the COE detect a large percentage of traditional integration difficulties. Moreover, the process of integration is largely automated as a byproduct of the installation tools themselves. By its very nature, the DII COE process pushes integration responsibilities further down to the original developer than is done with more traditional approaches.

- *Configuration Management is simplified.* One way that the COE process simplifies configuration management is by using segment descriptors that allow dependencies on, or conflicts with, other segments to be expressed. It then becomes possible to express the requirement for a top-level functional capability (e.g., a tool for editing an Air Tasking Order) and then recursively traverse a dependency tree to identify all required segments for the desired capability.

These benefits apply equally well to UNIX and NT environments and are in fact not dependent upon the underlying operating system.

The DII COE is a *superset* of capabilities. It contains far more functionality than would ever be installed on a single platform or even at a specific operational site. Thus, it is important to note and understand that just because a segment is part of the COE, it is not necessarily always present or required. Considerable flexibility is offered to customize the environment so that only the segments required to meet a specific mission-application need are present at runtime. This approach allows minimization of hardware resources required to support a COE-based system.

To illustrate the point, consider an example. The COE contains a service for displaying maps. However, some C4I operators in command centers only need to read and review message traffic and do not need or want to view a tactical display. Logistics operators using GCSS do not need to see the tactical picture at all and may only desire to see a map when planning transportation routes. For such operators it is not necessary at runtime to have the extra memory and performance overhead of the segments that generate cartographic displays.

Understanding the concept of a segment is fundamental to understanding and using the DII COE. It is, however, only the starting point. Given the background on how COE-based systems are packaged, it is now time to understand the internal structure of the DII COE.

2.1.1 COE Taxonomy

Segments that comprise the COE can be categorized in several ways. The original GCCS COE was subdivided into 19 functional areas and was organized largely by technologies employed such as network, database, and Mapping, Charting, Geodesy, and Imaging (MCG&I). Working groups were established for each of the 19 functional areas to consolidate operational requirements from each of the services/agencies and to evaluate and recommend candidate modules as core components. This taxonomy was initially successful and led to several early successes. However, the large number of working groups defined by this taxonomy quickly became unwieldy and communication within and between working groups became infeasible.

The DISA COE Design Working Group revisited the COE taxonomy as part of the effort to expand the GCCS COE into a DII COE. The present taxonomy consists of two layers: Infrastructure Services and

Common Support Applications[11]. These two layers are described in more detail in the *Architectural Design Document for the Defense Information Infrastructure (DII) Common Operating Environment (COE)*, and summarized in Figure 2-1. While encompassing the same functionality as the original 19 functional areas, this taxonomy approaches the problem from an architectural perspective rather than functional, and greatly reduces the communications burden in and between working groups. Figure 2-1 will be updated to include other functional areas as appropriate as the COE is extended to other mission domains. It has been updated since the *Architectural Design Document* to extend it for logistics support and to include a Web Server. This server is provided to allow access to COE-based applications from a Web browser. A later chapter in the *I&RTS* describes the COE Web in more detail.

The difference between Infrastructure Services and Common Support Applications is the difference between *data* and *information* (i.e., processed data). It is the difference between *exchanging* data and *sharing* data. Infrastructure Services provide low-level tools for data exchange. These services provide the architectural framework for managing and distributing the flow of data throughout the system. Example services include Transmission Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol (UDP) protocols, DCE, and CORBA. The achievement of effective data sharing requires use of all the COE services, especially those provided by the Shared Data Environment (SHADE). Subsection 2.1.2.5 describes SHADE in more detail.

Common Support Applications, on the other hand, provide the architectural framework for managing and disseminating information flow throughout the system, and for sharing information among applications. This level contains facilities for processing and displaying common data formats and for information integration and visualization. Services in this layer tend to be mission-domain specific. Examples include generation and dissemination of mission-relevant alerts, and word processing support.

Figure 2-1 also shows that there is a relationship between the service provided and whether it is typically provided by a COTS product or a GOTS product. The DII COE uses COTS whenever possible, in keeping with DOD directives. Infrastructure Services are normally provided by COTS solutions because they are closely tied to underlying vendor products such as the operating system. Common Support Applications, because the services they provide are closely related to mission applications, tend to be provided by GOTS solutions. In some cases, especially in the Office Automation area, services may include COTS solutions.

Selection of software modules that fulfill these COE component responsibilities is an ongoing task as is the evolutionary nature of the DII COE. Changes are made to further populate the COE, to optimize selected components, or to extend the COE to meet requirements from other mission domains. Even though the process is evolutionary, the COE preserves backwards compatibility so that mission applications are not abandoned just because there is an update of the COE. Refer to the appropriate API, User's Guide, and system release documents for detailed information on the components currently selected for the COE.

2.1.2 COE Architecture

Figure 2-2 is a simplified diagram that illustrates the various levels within the DII COE and the relationship between the COE, component segments, mission-application segments, and SHADE. As can be seen, the COE encompasses APIs, GOTS and COTS software, the operating system, windowing software, standards (*TAFIM*), and specifications (*User Interface Specification*, *I&RTS*, etc.). Physical databases are also considered to be part of the COE[12], including the software (such as the RDBMS), which accesses and

[11] The concepts of a *COE kernel* and *SHADE* are presented later in this section. Both of these concepts should be viewed as subsets of the Infrastructure Services and Common Support Applications layers. The COE kernel is a limited subset of the Infrastructure Services that is required on every platform regardless of how it will be used. SHADE is a subset of Infrastructure Services and Common Support Applications that deals with database issues. It is frequently useful to discuss the kernel and SHADE as separate entities, but their functionality is fully contained within the two layer taxonomy discussed in this section.

[12] In previous COE releases, physical databases were considered outside the scope of the COE although the database software was inside the COE. The programmatic decision to temporarily exclude physical

manages the data. SHADE is an integral part of the DII COE, and encompasses databases and related software as noted in the diagram. SHADE and each of the layers are described in more detail below. A Developer's Toolkit is also provided in the COE as shown in Figure 2-2.

Figure 2-2 is a generic diagram intended only to show relationships. The labeled boxes in the figure are not intended to be exhaustive, but are representative services because (a) otherwise the diagram would be needlessly complicated, and (b) the COE is evolving to include other segments to support new mission domains. The services shown are representative, but the structure and principles discussed are the same across all mission domains.

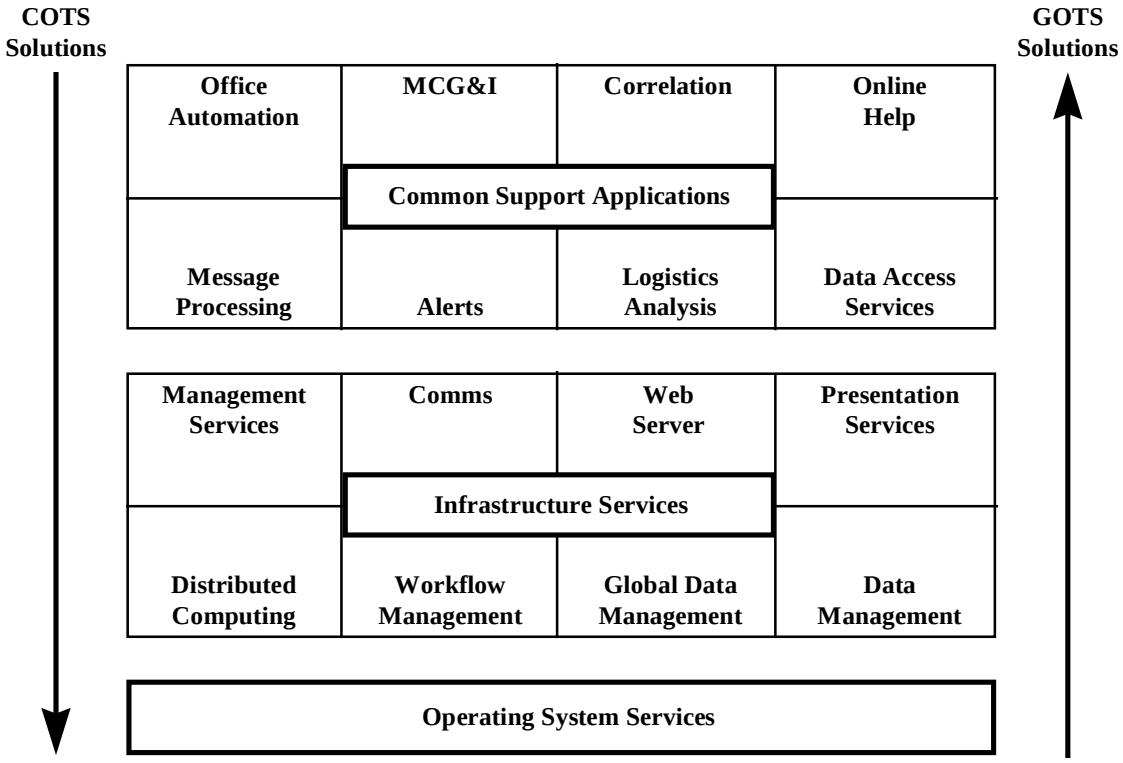


Figure 2-1: DII COE Services

databases was made in order to concentrate on the services that would more directly support mission applications. Databases are now included in the DII COE as part of the SHADE.

To use a hardware analogy, the COE is a collection of building blocks that form a software "backplane." Segments "plug" into the COE just as circuit cards plug into a hardware backplane. The blocks containing the operating system and windowing environment are akin to a power supply because they contain the software which "powers" the rest of the system. The segments labeled as COE-component segments are equivalent to already built boards such as Central Processing Unit (CPU) or memory cards. Some of them are required (e.g., CPU) while others are optional (e.g., a specialized communications interface card) depending upon how the system being built will be used. The blocks in Figure 2-2 labeled as mission application areas are composed of one or more mission-application segments. These segments are equivalent to adding custom circuit cards to the backplane to make the system suitable for one purpose or another.

The API layer shown in Figure 2-2 defines how other segments may connect to the backplane and utilize the "power supply" or other "circuit cards." This is analogous to a hardware schematic diagram that indicates how to build a circuit card that will properly plug into the backplane. The figure also implies that APIs are the only avenue for accessing services provided by the COE. This is true for all COE software and all layers, including COTS software. However, the COE does not create an additional layer on top of the COTS software. These components may be accessed directly using vendor-supplied APIs for these commercial products as long as such usage does not circumvent the intended COE architecture. For example, the COE includes a POSIX-compliant operating system. Some vendors provide non-POSIX compliant extensions to the operating system services. Use of such extensions, even though they are readily available through vendor-supplied APIs, is not allowed because such usage violates the intended COE architecture.

This hardware analogy can be extended to the SHADE portion of the COE, but with some significant distinctions. Within this conceptual model, the Database Management System (DBMS) functions as the COE's disk controller and disk drives. The applications' databases can be equated to directories or partitions on the drives accessed through the DBMS "disk controller." Data objects belonging to each database then can be considered as files within those "directories."

This analogy is critical to understanding the modularity limitations for databases within the COE. One can replace most peripherals or circuit cards without any side effects just as one can replace mission applications without losing information. However one cannot put in a larger disk drive, or change from one type of controller to another, without losing the data on the disk. While upgrading mission applications is like swapping circuit cards, upgrading databases is like rebuilding a disk or directory structure. Instead of replacing a component, one must save and then restore the files on the disk. Proper design of COE/SHADE databases must provide the ability to perform field upgrades without the loss of any data.

COE/SHADE databases are divided among segments as are mission applications, but with a different focus. Mission applications are segmented based on their functionality. Databases are segmented functionally by the subject areas of the mission applications they support. Mission applications are functional modules; databases are information modules.

The precise configuration of COTS products used in the COE is placed under strict configuration control. This is necessary because configurable items such as the amount of shared memory or swap space must be known and carefully controlled in order for other components in the COE to operate properly. For this reason, COTS products are assigned a version number in addition to the vendor-supplied version number so as to be able to track and manage configuration changes. Databases are also assigned version numbers because their configurations must be controlled since the data content may change from release to release, or the database schema may change.

A fundamental principle throughout the COE is that segments are not allowed to directly modify any resource "owned" by another segment. This includes files, directories, modifications to the operating system, and modification to windowing environment resources. Instead, the COE provides tools through which a segment can request extensions to the base environment. The importance of this principle cannot be overemphasized because environmental interactions between software components are a primary reason for difficulties at integration time. By providing software tools that arbitrate requests to extend the

environment, integration can be largely automated and potential problem areas can be automatically identified.

For example, the COE predefines a set of ports in the UNIX `/etc/services` file. Some segments may need to add their own port definitions, but this will create conflicts if the port definitions are the same as those defined by the COE or another segment. To identify and prevent such conflicts, segments issue a request to the COE (see Chapter 5 for how this is done) to add their port definitions. This process is called *environment extension* because a segment is modifying the predefined environment by extension, not through replacement or deletion.

COE-component segments shown in Figure 2-2 are typically designed to be servers, although some are provided as libraries to be linked with an application segment. Note that in practice such segments will often operate in both a client and server mode. For example, a track management segment is a server for clients that need to retrieve the current latitude/longitude location of a platform. But the track manager itself is a client to a communications server that initially receives track-related reports from sensors or other sources. Refer to the *Architectural Design Document for the Defense Information Infrastructure (DII) Common Operating Environment (COE)* document for more detailed discussion of how COE-component segments are designed and interact. For purposes of the present discussion, it is sufficient to view COE segments as servers that are accessible through APIs.

2.1.2.1 COE Kernel

The COE will normally make available a large number of segments, not all of which are required for every application. The *COE kernel* is the minimal set of software required on every platform regardless of how the platform will be used. The COE kernel[13] components are shown in Figure 2-2 and include the Operating System and Windowing Services and a collection of other services that properly belong in the Infrastructure Services Layer.

[13] The kernel includes both COTS (e.g., operating system) and GOTS software. The other COE layers also contain COTS software. Contact the DII Engineering Office for information on responsibility for obtaining licenses for COTS products within the COE and kernel, and for which COTS product DISA will distribute.

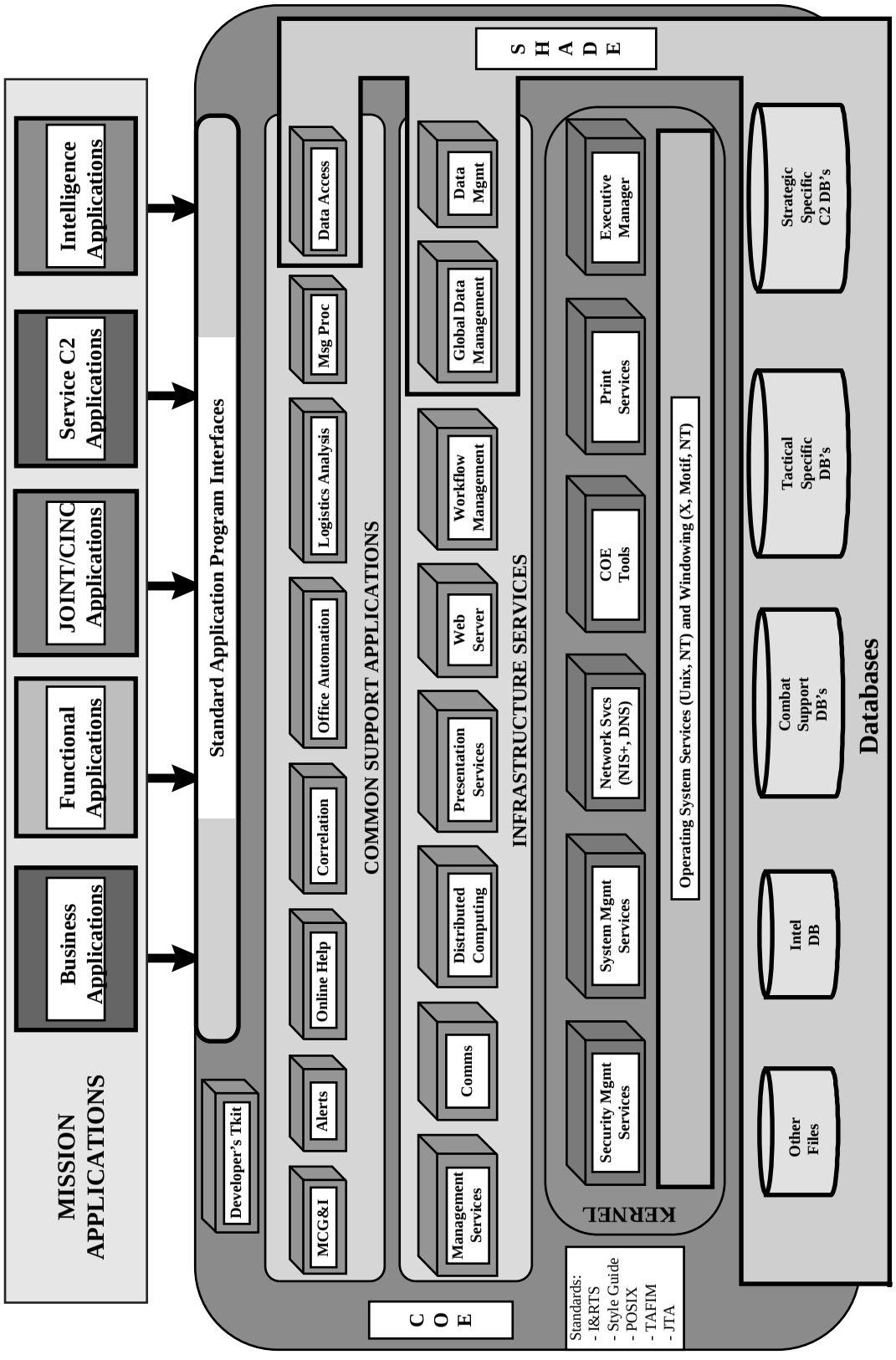


Figure 2-2: DII COE Architecture

A COE kernel will always contain the operating system and windowing environment, but it will normally include six other features:

1. a basic System Administration function,
2. a basic Security Administration function,
3. an Executive Manager function (e.g., a desktop GUI such as Windows NT or Common Desktop Environment [CDE]),
4. a template for creating privileged operator login accounts,
5. a template for creating non-privileged operator login accounts, and
6. COE tools for segment installation.

The System Administration segment is required because it contains the software necessary to perform basic system administration tasks such as user account and profile management. The Security Administration segment is required because the security administrator uses it to enforce system security policy. The operating system and other COE components provide security policy enforcement. Segments loaded later may provide additional system and security administration capabilities, but the minimum capabilities for security enforcement and security administration are in the kernel.

The Executive Manager component of the kernel is required because it is the interface through which an operator issues commands to the system. The Executive Manager is an icon-and-menu-driven desktop interface, not a command-line interface. The templates included in the COE kernel are used to define the basic runtime environment context that an operator inherits when he logs in (which processes to run in the background, which environment variables are defined, etc.). The COE tools within the kernel allow other segments to be installed and enforce critical COE principles. The COE kernel assures that every platform in the system operates and behaves in a consistent manner and that every platform begins with the same environment.

2.1.2.2 Infrastructure Services

Infrastructure Services are largely independent of any particular application. Within the Infrastructure Services layer, Management Services include network, system, and security administration. Communications Services provide facilities for receiving data external to the system and for sending data out of the system. Distributed Computing Services provide the infrastructure necessary to achieve true distributed processing in a client/server environment. Presentation Services are responsible for direct interaction with the human whether that be through windows, icons, menus, or multimedia. Data Management Services include relational database management as well as file management in a distributed environment. Workflow and Global Data Management Services are oriented towards managing logistics data (e.g., parts inventory, work in process). Note that Data Management Services and Global Data Management Services are part of SHADE.

2.1.2.3 Common Support Applications

Unlike Infrastructure Services, Common Support Applications tend to be much more specific to a particular mission domain. The Alerts Service is responsible for routing and managing alert messages throughout the system whether the alert is an “out of paper” message to a systems administrator or an “incoming missile” alert to a watch operator. The Correlation Service is responsible for maintaining a consistent view of the battlespace by correlating information from sensors or other sources that indicate the disposition of platforms of interest. MCG&I Services handle display of National Imagery and Mapping Agency (NIMA) maps or other products, and imagery received from various sources. Message Processing Services handle parsing and distribution of military-format messages. Office Automation Services handle word processing, spreadsheet, briefing support, electronic mail, World-Wide-Web browsers, and other related functions. (Browsers are in the Common Support Applications layer, but Web Servers fall within the Infrastructure Services layer.) Logistics Analysis contains common functions, such as Pert charts, for analyzing and displaying logistics-related information. Online Help Services provide applications with a uniform

technique for displaying context-sensitive help. Finally, Data Access Services are part of SHADE and provide applications with common data-access methods procedures, and tools.

2.1.2.4 COE Developer Toolkits

Since the COE is not a system but a foundation on which systems are built, the COE contains a collection of developer toolkits to assist the developer in creating mission-application software. This is illustrated in Figure 2-2 in the block labeled Developer's Toolkit. However, the toolkits are required only during software development, not during runtime at an operational site. Therefore, developer toolkits are shown as part of the COE, but outside the Infrastructure Services and Common Support Applications layers. They are obtained from DISA separate from an actual installable system.

The COE developer toolkits contain libraries of APIs and a collection of tools to assist in the segmentation process. An overview of the software development process is presented in the next chapter. Appendix C provides an overview of the COE developer tools (and lists some COE runtime tools). Refer to the appropriate *DII COE Programmer's Guides* for detailed information on the APIs and segmentation tools.

2.1.2.5 SHADE

SHADE is an important addition to this version of the *I&RTS*. Its purpose is to provide the data "missing piece" for the DII COE. The present subsection provides an overview of SHADE and describes how it fits into the overall DII COE. A later chapter will cover SHADE and database topics in much more technical detail and depth.

Present systems are not truly interoperable because of inconsistency in algorithms, but also because data management across systems and operational sites has led to data redundancy and inconsistencies. Moreover, even when data is consistent across systems, it is not presently structured so as to be shareable. The SHADE approach is to provide the architectural structure to solve the data sharing problem that in turn guarantees data consistency, eliminates redundancy[14], and promotes true data interoperability and sharing. The SHADE goal is to allow any authorized user from any authorized workstation to locate, access, and integrate shared and synchronized data. This is in keeping with the DISA vision of an integrated global environment that allows warriors to perform "Any Mission, Any Time, Any Where."

SHADE is both a strategy for data sharing and the mechanisms to achieve it. SHADE is an integral part of the DII COE, but it must also bridge the gap between COE-based systems and legacy non-COE systems because it must provide mechanisms for accessing large databases that are still on legacy mainframes. SHADE provides COE-component segments in both the Infrastructure Services and Common Support Applications layers to accomplish this task. SHADE includes the required data-access architectures, data sharing methodology, reusable software and data components, and guidelines and standards for the development and migration of systems that meet the user's requirements for timely, accurate, and reliable data.

The SHADE components of Figure 2-2 are expanded upon in Figure 2-3 to show the architecture from a data management perspective. From a process point of view (top part of the diagram), SHADE includes

[14] Databases are often deliberately replicated in actual practice for performance reasons. The term "redundant data" is used when the same data is captured by different systems and stored in different databases. For example, the friend/foe status of a particular country might be entered into two systems where each system must maintain and keep the data current. By contrast, when intentional replication is used, the friend/foe status is captured and maintained by one system and provided to another for its use. SHADE may not rule out multiple copies of the same data but it does manage the duplication to ensure that all databases are kept synchronized. Present systems often do not employ effective mechanisms for data replication, leading directly to significant interoperability problems. SHADE does eliminate redundancy *between* systems because, for performance reasons, it replicates and manages duplication *across* systems to ensure data consistency.

tools for validating database segments and a repository for data reuse. Metadata Management is at the top layer between the mission applications and data-access methods. This layer is among the more challenging aspects of SHADE because it requires standardization across the joint community. The Shared Data Access layer provides services for locating and retrieving the desired data. This layer also manages data replication and distribution to ensure that all databases are kept closely synchronized. Data security is also provided in this layer.

The Physical Data Management layer is provided by commercial products and is initially organized as relational databases. (Migration to include other database management technologies such as object-oriented or object relational will be achieved as requirements emerge and technology matures.) SHADE physical data management services may also include document retrieval, image management, engineering drawings, or other specialized storage and retrieval technologies where appropriate. The databases may be distributed across the network, and may in fact be distributed among geographical sites.

Figure 2-3 shows three types of database segments according to their scope and how they are shared. The three types are Unique, Shared, and Universal.

Unique database segments are those which are typically used by only one application or are under the configuration control of the segment sponsor. Unique data may be shared between applications, but the usage is restricted to a single mission domain. An example of a Unique database segment is a configuration table that an application reads at initialization time. Such a table would not normally be used by other applications. This example also demonstrates that Unique database segments may frequently be represented by a flat file or similar structure rather than a true database.

Shared database segments support the information requirements of multiple applications or across multiple database segments. Shared database segments are typically mission-or-functionally-oriented, and are generally specific to a limited number of mission domains. Because they affect multiple applications that will likely span services or functional areas, Shared database segments must be under joint configuration control. An example of such a database segment is a database of logistics drawings for military hardware. Such data spans multiple services, it is used for different purposes (e.g., ordering, inventory control, maintenance) and hence spans multiple applications, but it is generally limited in scope to the logistics community. Another example is a segment containing invoice information that is required by both the finance and procurement communities.

Universal database segments represent the other extreme of "shareability." Universal database segments reflect a need for identical data in diverse areas, are used by many applications, and span multiple mission domains. Universal database segments usually have no dependency on any other segment (except the DBMS segment) and frequently consist of a small number of tables and elements. A common type is reference or lookup tables. An example is a database of country-code abbreviations. A larger example would be the equivalent of "Jane's Data" with characteristics and performance data concerning weapons, aircraft, ships, and communications systems. Universal database segments are under stricter configuration control and require DISA and DOD Data Administration coordination.

The three database segment types are listed in increasing order of scope and "shareability." That is, Unique is limited in scope and therefore unlikely to be shared by many applications, while Universal is very broad in scope and *must* be shared across applications in order to promote true interoperability. There is no physical difference in the database segments, but the level of configuration management increases due to the wider impact changes would have on operational systems that use the database segments.

2.1.2.6 COE "Plug and Play"

The DII COE is structured as a "plug and play" architecture. The key to the "plug and play" design is conformance to the COE through the rules detailed in this document and through using only the published APIs for accessing COE services. There is considerable danger in using unpublished, "private" APIs, or APIs from legacy systems, because there is no guarantee that interfaces used in this fashion will remain the

same or even exist in subsequent releases. This is also generally true of COTS products and the risks are the same.

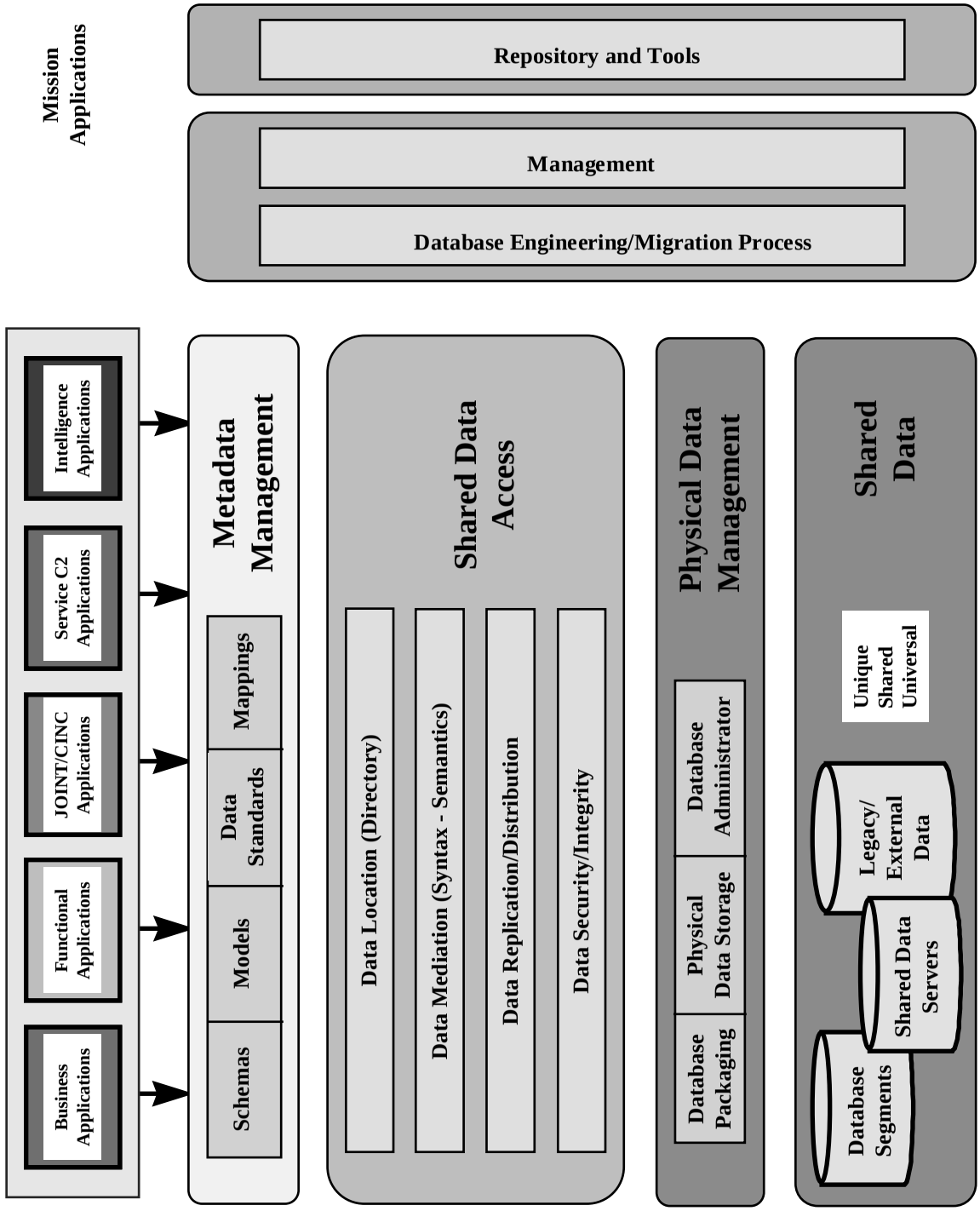


Figure 2-3: SHADE Data Architecture

Discussion of the COE as a "plug and play" architecture is not intended to trivialize the effort that may be required to develop and integrate a segment into the COE. Migration of existing legacy systems to the COE is conceptually straightforward but may require considerable effort due to the requirement to switch to a different set of building blocks. That is, the effort may not be so much in adjusting to a new architectural concept but in adjusting code to use a different set of APIs. The "plug and play" paradigm is a good conceptual model because it clearly conveys the goal and the simplicity that most segment developers will encounter.

2.1.3 COE Configuration Definitions

A COE-based system will consist of a large number of segments. It is neither desirable nor feasible to install all segments on all platforms. Some segments need to be installed on one platform but not another because of the role that the platform will play in the overall system. For example, systems will often dedicate one or more platforms with large-capacity disk drives to be configured as database servers. Workstations that operators use, client workstations, will not have large enough drives to handle the database storage requirements. Therefore, the database server software should be loaded on the database server but not the client platform. The COE kernel is required on every platform, but additional segments are dependent upon how the platform will be used.

The COE includes the ability to create configuration definitions that define which segments are to be loaded on which platforms. A *configuration definition*[15] is a hierarchy that defines collections of segments that are grouped together for installation convenience. For example, it is more convenient for an installer to indicate that a platform is to act as a database server (a configuration definition) or used as an intelligence analyst workstation (another configuration definition) than to manually and individually select all of the segments that need to be installed. The COE is designed so that a site may install predefined configuration definitions or can customize the installation to suit site-specific requirements.

A configuration definition is organized into folders, configurations, and bundles. Figure 2-4 uses an example from the GCCS system to show the relationship between each of these terms, and to illustrate the flexibility in predefining and managing software installations. The example shows how the GCCS system *could* be organized into configuration definitions, but not how GCCS *must* be organized. The example is not intended to convey that platforms must be dedicated to a single, specific function. As long as there are no segment conflicts, a platform may be configured to support multiple missions and thus achieve the goal of "any platform for any function." The example is intended only as an aid to understanding how configuration definitions may be constructed.

The objective of the example shown in Figure 2-4 is to install identical database servers in Intelligence centers at two GCCS sites: a Commander, Joint Task Force (CJTF) and a Commander-in-Chief Headquarters (CINCHQ). In this simplified example, both Intelligence centers use imagery applications, but the Intel center at the CINCHQ has access to hardware for capturing images, while the one at the CJTF does not.

For simplicity, the database server is to consist of 4 segments: a segment for creating database backups (`Bkup`), an ad hoc query application (`AdHocQ`), a presentation package (`Forms`), and a patch (`Patch1`). The imagery software is to consist of an application for creating briefs (`Brief`), an application for capturing images (`Capture`), and an application for converting images from one format to another (`Convert`).

[15] The term *Configuration Definition* replaces the term *variant* in previous *I&RTS* releases. The concepts are exactly the same except that variant has the negative connotation of implying a "deviation." Further, the Configuration Definition concept is more refined in this *I&RTS* version in its decomposition into folders, configurations, and bundles.

A *configuration definition file* is a file that describes the hierarchy and relationships among folders, configurations, bundles, and segments that comprise a distribution. A *distribution*[16] is the physical media used to install DII-compliant segments (e.g., Digital Audio Tape [DAT] tape, 8mm tape, Compact Disc Read Only Memory [CDROM]). A single distribution may span multiple media of the same type (e.g., several DAT tapes, several CDROMs). A configuration definition file is used to generate the table of contents for what is contained in the distribution.

In addition to being physical media, another useful way to think of a distribution is as a high-level division that can be used to distinguish between systems (e.g., GCSS, GCCS, ECPN), as shown in this example. The example would work equally well by defining the desired distributions one level lower in the tree and thus place responsibility for site configurations on a manager responsible for the site, rather than on a manager responsible for GCCS configurations at all sites. A distribution is not sufficiently detailed to permit the actual installation of any software since it must be decomposed further to the level of an actual platform.

A *folder*, likewise, is a non-installable list of one or more folders, configurations, or bundles. Folders are used for organizational and display purposes only. A folder is not directly installable because it is organized at a level that spans multiple platforms and perhaps even multiple sites. In this example, the GCCS distribution media is composed of multiple folders at the top level representing geographically dispersed sites. The next lower level of folders is contained within a single site.

A *configuration* is a list of bundles and/or segments that can be installed on a single machine. Configurations are mutually exclusive. That is, only one configuration can be installed on a single machine because there may be conflicts within the segments that comprise two different configurations. In the example, the `Intel` folder for the CJTF site contains two configurations for platforms: a database server configuration (`DBServer`) and an imagery configuration (`Imagery1`). A particular platform may be loaded with a `DBServer` or an `Imagery1` configuration, but not both. This does not mean that imagery applications cannot reside on a database server. It only means that in this example, an engineering decision was made to prevent it from happening because of potential resource conflicts between the two configurations. If it were actually desirable to combine the applications in practice, a configuration could be defined which contained database server and imagery bundles. Or, desired segments could be selected individually for loading onto the platform.

[16] The *distribution* term is a POSIX concept. It has been modified slightly in the *I&RTS* to include segments.

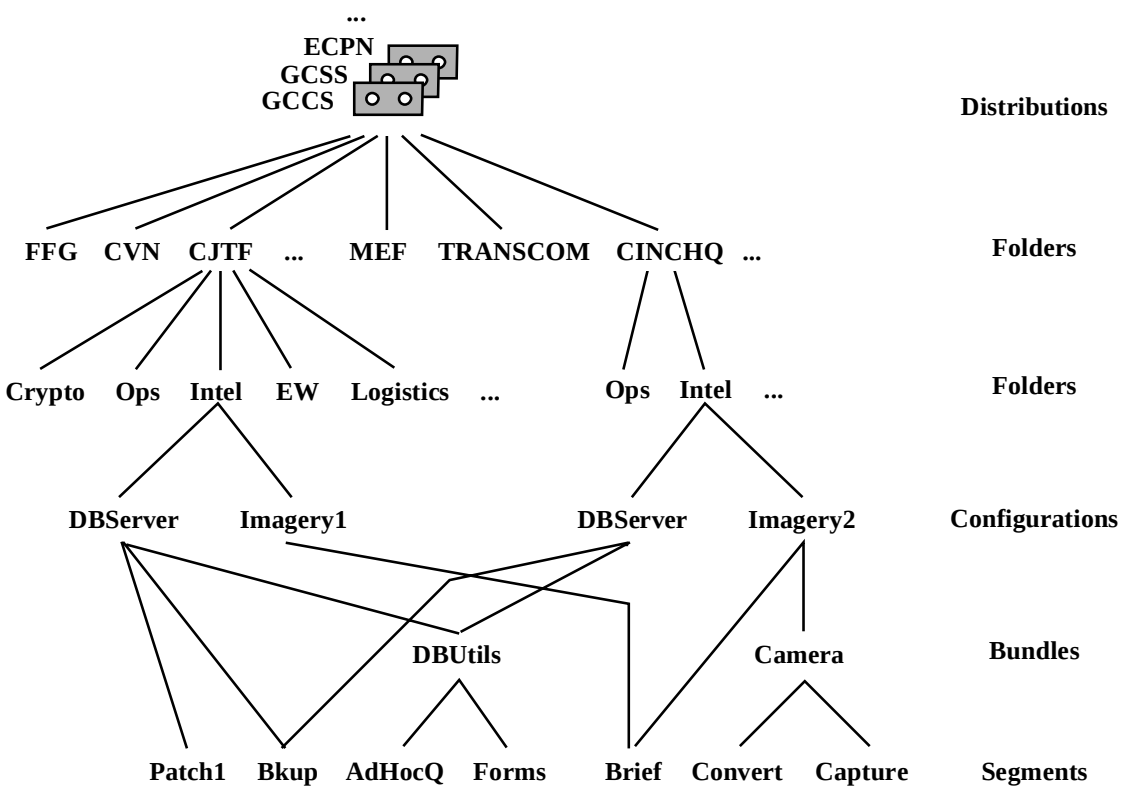


Figure 2-4: Configuration Definitions

A *bundle*[17] is a list of other installable bundles and/or segments. For brevity, Figure 2-4 does not show any bundles that contain other bundles. A bundle is directly installable, even if it contains further bundle definitions because the segments that comprise the bundle are checked when the bundle is created to verify that they do not conflict with one another. In the example shown, there are two bundles: `DBUtils` and `Camera`. The `DBUtils` bundle is used at both the `CJTF` and `CINCHQ` sites, but `Camera` is only used at `CINCHQ`.

There are several things to note about this example.

1. At installation time, the installer can use the installer tool to select a configuration and all of the appropriate segments will automatically be installed. The installer may also choose to decompose the configuration to look at individual bundles and segments and install them individually instead.

2. Configurations, bundles, and segments may be selected and installed directly without further selection on the part of the installer. Folders cannot; the installer must select some lower level in the hierarchy.

3. Folders may participate in multiple distributions or other folders; configurations may participate in multiple folders; bundles may participate in multiple configurations; and segments may participate in multiple bundles or configurations. Multiple participation is subject to the constraint that segments within a configuration or bundle cannot contain conflicting segments.

4. Configuration definitions are optional. They are provided as a convenience only. Also, it is possible to skip any of the levels in the configuration definition except for the lowest level (i.e., segments).

[17] The *bundle* concept is from POSIX, but has been modified slightly in the *I&RTS* to include segments.

5. If a segment is selected twice either on purpose or as a result of how the configuration definition is constructed, it is actually installed only once.

6. Care should be taken in creating configuration definitions. It is wise to keep classified segments separate to avoid security management problems.

The same media can be used to load any platform regardless of which site or in which space the platform is located; however, during the installation process, only that portion of the configuration definition required for a particular platform is actually loaded. The COE kernel is a required member of every distribution.

There are several advantages to configuration definitions:

- From a configuration management and security perspective, only one set of distribution media needs to be controlled. All software and data that are needed for the installation are contained on the media.

- From an installation perspective, the site installer only has one set of distribution media to worry about regardless of platform use or hardware type. (The COE tools allow segments for multiple platforms to exist on the same physical distribution media. At installation time, the software determines the platform type and then makes available for selection only those segments that can execute on the platform.)

- From a system design perspective, the ability to create configuration definitions allows the flexibility of loading and executing only that software which is required to support a particular mission requirement.

2.1.4 DII Compliance

The degree to which "plug and play" is possible is highly dependent upon the degree to which segments are DII-compliant. DII compliance is defined to be an integer value that measures

- the degree to which a segment or system achieves conformance with the rules, standards, and specifications identified by the COE,
- the degree to which the segment or system is suitable for integration with the DII COE reference implementation, and
- the degree to which the segment or system makes use of COE services.

Appendix B contains a detailed checklist for areas where compliance is mandatory and an additional checklist for areas where compliance will be required in the future but are optional at present. The compliance level for a segment is determined by answering "True," "False," or "N/A" for each question in the checklist. The Category 1 (see below) compliance level assigned is the highest numbered level for which there are not "False" replies. The COE provides a suite of tools, described in Appendix C, which validate COE conformance.

By its very nature, an exhaustive list of "do's and don'ts" is not possible. DII compliance must be guided by overarching principles with checklists and tools to aid in detecting as many problem areas as possible. Full DII compliance embodies the following principles:

1. Segments shall comply with the guidelines, specifications, and standards defined in the *I&RTS*, the *User Interface Specification*, *DII Software Quality Compliance Plan*, and related documents such as the *JTA*.

2. Software and data shall be structured in segment format. Of necessity, COTS components of the COE kernel are exempted from this requirement. Segment format is described more fully in Chapter 5.

3. Segments shall be registered and submitted to the online library. The registration process is described in Appendix E while submission of segments to the online library is described in Chapter 10 and Appendix D.

4. Segments shall be validated with the `VerifySeg` tool prior to submission, and shall successfully pass the `VerifySeg` tool with no errors. An annotated listing of the `VerifySeg` tool output shall be submitted with each segment release.

5. Segments shall be loaded and tested in the COE environment prior to submission. Segment developers are responsible for testing their segment within the full COE kernel and with all COE-component segments that they depend upon. There is no requirement to include mission-application segments in the test for which there is no dependency.

6. Segments shall fully specify dependencies, conflicts, and required resources through the appropriate segment descriptors defined in Chapter 5.

7. Segments shall be designed to be removable and tested to confirm that they can be successfully removed from the system. Some segments, especially COE components, are designed to be "permanent" but even these must be removable when a later segment release supersedes the current one.

8. Segments shall access COE components only through APIs published by DISA and segments shall not duplicate functionality contained within the COE. There is no requirement to integrate to COE functionality not required by the segment, but note that some segments may have an implied dependency on other segments.

9. Segments shall not modify the environment or any files it does not own except through environment extension files or through use of the installation tools provided by the COE.

The DII COE defines four areas in which compliance is measured, shown in Figure 2-5, called compliance *categories*. Within a specific category, a segment is assigned an integer value, called the compliance *level*, which is a measure of the degree to which a segment is compliant within that category. The DII COE takes this approach because it is especially useful in developing migration strategies for legacy systems. Compliance categories indicate the broad area in which a segment must be improved while compliance levels express the degree to which the segment meets COE objectives within that category.

The four DII compliance categories are:

Category 1: Runtime Environment. This category measures how well the proposed software fits within the COE executing environment, and the degree to which the software reuses COE components. It is an assessment of whether or not the software will "run" when loaded on a COE platform, and whether or not it will interfere with other segments. This category is closely related to, and is a way of measuring, interoperability.

Category 2: Style Guide. This category measures how well the proposed software operates from a "look and feel" perspective. It is an assessment of how consistent the overall system will appear to the end user. It is important that the resulting COE-based system appear seamless and consistent to minimize training and maintenance costs.

Category 3: Architectural Compatibility. This category measures how well the proposed software fits within the COE architecture (client/server architecture, DCE infrastructure, CDE desktop, etc.). It is an assessment of the software's potential longevity as the COE evolves. It does *not* imply that all software must be based on client/server or Remote Procedure Call (RPC) techniques. It simply means that a reasonable design choice has been made given that the specific architectural characteristics of the COE reference implementation.

Category 4: Software Quality. This category measures traditional software metrics (lines of code, McCabe complexity metric, etc.). It is an assessment of program risk and software maturity.

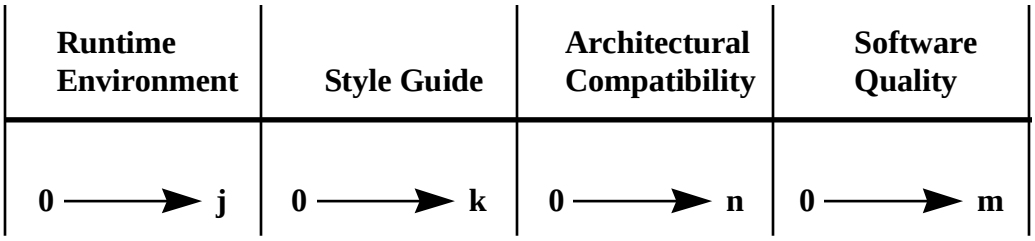


Figure 2-5: DII Compliance Categories and Levels

Note: While there are four compliance categories, style-related items are included within the *I&RTS* checklist. Specifications within the *User Interface Specification* are mapped to these items at the appropriate compliance level where they are included. For example, Category 1 (Runtime Environment) Level 5 compliance requires adherence to the "look and feel" of the native GUI. The *User Interface Specification* contains a checklist for verifying that a segment conforms to the native GUI.

These four categories attempt to quantitatively answer the following questions about a proposed addition to the system:

- (Category 1: Runtime Environment) Can the proposed software be added to the system? Will it adversely affect system interoperability?
- (Category 2: Style Guide) Is the proposed software user-friendly? Will it make the system appear seamless to an end user?
- (Category 3: Architectural Compatibility) Is the proposed software architecturally sound and in line with where the COE is going? Will technology advances quickly obsolete the proposed software?
- (Category 4: Software Quality) What is the program risk? Will significant program expenditures be required for life-cycle maintenance of the product?

The principles and techniques described in the remainder of this subsection apply to each of the compliance categories. However, only the compliance levels for the Runtime Environment Category will be discussed any further.

The COE defines eight progressively deeper levels of integration for the Runtime Environment Category. These levels are directly tied to the degree of interoperability achieved as is described in subsection 2.1.5. Note that levels 1-3 are "interfacing" with the COE, not true integration. True integration begins at level 4.

Level 1: Standards Compliance Level. A superficial level in which the proposed capabilities share only a common set of COTS standards. Sharing of data is undisciplined and minimal software reuse exists beyond the COTS. Level 1 may, but is not guaranteed to, allow simultaneous execution of the two systems.

Level 2: Network Compliance Level. Two capabilities coexist on the same LAN but on different CPUs. Limited data sharing is possible. If common user interface standards are used, applications on the LAN may have a common appearance to the user.

Level 3: Platform Compliance Level. Environmental conflicts have been resolved so that two applications may reside on the same LAN, share data, and coexist on the same platform as COE-based software. The COE kernel, or its equivalent, must reside on the platform. Segmenting may not have been performed, but some COE components may be reused. Applications do not use COE services (except for kernel services if the COE kernel is loaded) and are not necessarily interoperable.

Level 4: Bootstrap Compliance Level. All applications are in segment format and share the COE kernel. Segment formatting allows automatic checking for certain types of application conflicts. Use of COE services is not achieved and users may require separate login accounts to switch between applications.

Level 5: Minimal DII Compliance Level. All segments share the same COE kernel and functionality is available via the Executive Manager. Boot, background, session, and local processes are specified through the appropriate segment descriptors. (See Chapter 5 for a description of the types of processes.) Segments adhere to the basic “look and feel” of the native GUI, as defined in the *User Interface Specification*. Segments are registered and available through the online library. Applications appear integrated to the user, but there may be duplication of functionality and full interoperability is not guaranteed. Segments may be successfully installed and removed through the COE installation tools. Database segments are identified as unique or sharable according to their potential for sharing.

Level 6: Intermediate DII Compliance Level. Segments utilize existing account groups, and reuse one or more COE-component segments. Minor documented differences may exist between the *User Interface Specification* and the segment's GUI implementation. Use of non-standard Structured Query Language (SQL) in database segments is documented and, where applicable, packaged in a separate database segment.

Level 7: Interoperable Compliance Level. Segments reuse COE-component segments to ensure interoperability. These include COE-provided communications interfaces, message parsers, database segments, track data elements, and logistics services. All access is through published APIs with documented use of few, if any, private APIs. Segments do not duplicate any functionality contained in COE-component segments. The data objects contained within a database segment are standardized according to DOD 8320 guidance.

Level 8: Full DII Compliance Level. Proposed new functionality is completely integrated into the system (e.g., makes maximum possible use of COE services) and is available via the Executive Manager. The segment is fully compliant with the *User Interface Specification* and uses only published public APIs. The segment does not duplicate any functionality contained elsewhere in the system whether as part of the COE or as part of another mission application or database segment.

Bootstrap Compliance (Level 4) is required before a segment may be submitted to DISA for evaluation as a prototype. Such segments will not be fielded nor accepted into the online library. At DISA's discretion, segments which meet the criteria for Minimal DII Compliance (Level 5) may be accepted into the online library, and installed at selected sites as prototypes for user evaluation and feedback. Such segments will not be accepted as fieldable products. Acceptance as an official DISA fieldable product requires demonstration of Interoperable Compliance (Level 7) and a migration strategy to Full DII Compliance (Level 8), unless the proposed segment is an interim product that is targeted to be phased out in the near term.

The compliance categories and levels defined here are a natural outcome of developing a reasonable approach to migrating legacy systems into the COE. The first step of Category 1, covered by Levels 1-4, is to ensure that systems do not destructively interfere with each other when located on the same LAN. Level 5 is sometimes called a "federation of systems" in that systems are still maintained as "stovepipes," but they can safely share common hardware platform resources. Levels 6-8 complete the approach by reducing functional duplication, promoting true data sharing, and making the system appear to the user as if it were developed as a single system. The last three levels represent varying degrees of integration from marginally acceptable (Level 6) to a truly integrated system (Level 8). All 8 levels represent a progressively deeper level of interoperability.

The same compliance levels apply to SHADE databases, as well. The majority of the SHADE issues in Levels 1-4 are concerned with proper use of the COTS database management systems' functionality and with not destructively accessing data belonging to other databases. At Level 5, a database must identify those components of its schema which are candidates for "sharing." Levels 6-8 reduce and then eliminate data sets that are redundant with information in shared and universal segments, including database design modifications and data migration and cleansing to provide interoperability in both data structure and content.

Compliance checking is done on a segment-by-segment basis according to the definitions given here and through the checklist approach in Appendix B. The categories and levels described here are independent of where the segment fits into the system. That is, the same definitions apply whether the segment is a COE-component segment or a mission-application segment. However, it is sometimes necessary to compute the compliance level of a collection of segments. This is called a *composite compliance level*. The remaining subsections below describe how to compute a composite compliance value for an arbitrary group of segments, for the COE itself, for a COE-based system, and for systems which contain both COE and non-COE based computing platforms. A composite value is required because otherwise a system is only as compliant as its least compliant segment and the least compliant segment may be in the COE[18] itself. Thus, the intent is to not penalize systems for non-compliant components in the COE itself.

Strictly speaking, discussion of DII compliance requires qualification with a category name, a compliance level, and whether compliance is being measured against a segment or a collection of segments. Thus, it is correct to say that a particular segment is Category 1, Level 4 compliant, but it could be confusing to omit the qualifier Category 1. Because of widespread usage in the COE community, when a category is not stated, Categories 1 and 2 are assumed.[19]

The *I&RTS* expressly uses integer values rather than decimals or percentages to state DII compliance. Expressing compliance as a percentage is both confusing and misleading. For example, to state that a segment is 85% Level 6 compliant can be interpreted in many ways. It could mean that 85% of the effort required to achieve Level 6 compliance has been achieved, or that 85% of the functionality in the system is 85% Level 6 compliant. However, it most likely means only that the segment successfully passes 85% of the Level 6 criteria in Appendix B. Because of the difficulty in precisely interpreting the intended meaning, only integer compliance values are allowed. Otherwise, it is difficult to quantitatively compare two segments or systems if both claim to be 85% Level 6 compliant.

[18] The COE reference implementation contains software contributed by legacy systems. It may not be cost effective to expend the effort to achieve full Level 8 compliance for some of these legacy contributions because they are going to eventually be phased out. In the interim, systems that use these segments should not be penalized for their lack of compliance.

[19] The *JTA* states a requirement for a minimum of Level 5 compliance as does OSD directive. In both cases, Category 1: Runtime Environment and Category 2: Style Guide are intended. The requirement is levied on individual segments, and on COE-based systems.

2.1.4.1 Compliance for an Arbitrary Group of Segments

Segments are often grouped together, as in a configuration definition. The composite compliance level for an arbitrary collection of segments is the compliance level for the *least* compliant segment.[20] For example, suppose a group of four segments have compliance levels of 5, 8, 3, and 8 respectively. Then the composite compliance level for this group of four segments is 3.

This approach to calculating composite compliance levels intentionally places a heavy penalty on groups that have segments with low compliance levels and gives no "credit" if there are segments with high compliance levels. An alternative approach would be to average the levels, but because compliance is a direct measure of interoperability and because artificially increasing the number of segments could have the misleading effect of boosting the apparent level of compliance, this approach was rejected.

2.1.4.2 Compliance for the DII COE

Calculating the compliance level for the COE itself requires computing the composite compliance level for 1) the COE kernel, and 2) for the Infrastructure Services and Common Support Applications layers. As described in subsection 2.1.4.1, the composite compliance level for each of these two groups of segments is the level of the least compliant segment in the group.

Let C_k be the composite compliance level of the COE kernel. Let C_c be the composite compliance level for the combined Infrastructure Services and Common Support Applications segments. Then the composite compliance level for the DII COE (C_{dii}) is given by the equation

$$C_{dii} = \mathbf{TRUNC}([C_k + C_c]/2)$$

where **TRUNC** means to truncate the result to an integer value.

Consider an example. Assume the kernel has three segments with compliance levels 6, 8, and 5. Assume there are four segments in the Infrastructure Services layer with compliance levels 8, 8, 7, and 4. Lastly, assume that there are seven segments in the Common Support Applications layer and all are level 8 compliant.

The composite compliance level for the combined Infrastructure Services and Common Support Applications segments is the compliance level of the least compliant segment (e.g., 4). Thus, the following gives the composite compliance level for the DII COE for this example:

$$C_k = 5$$
$$C_c = 4$$
$$C_{dii} = \mathbf{TRUNC}([5 + 4]/2) = \mathbf{TRUNC}[9/2] = 4.$$

2.1.4.3 Compliance for a COE-Based System

The composite compliance for a COE-based system is computed in a manner similar to that of computing the compliance for the DII COE. The approach is to compute the composite compliance level for the mission-application segments and then factor in the DII compliance. The computation here is valid only if every platform in the system is COE-based. If there is a mixture, refer to subsection 2.1.4.4.

Let C_{ma} be the composite compliance level of all the mission applications in the system. Let C_{dii} be the composite compliance level for the COE computed as described in subsection 2.1.4.2. In computing C_{dii},

[20] This is how the compliance level for an aggregate segment is measured. (See Chapter 5 for the definition of an aggregate segment.) The compliance level of an aggregate segment is the compliance level of the *least* compliant segment in the aggregate.

only those segments in the COE that are actually used in the resulting system are considered. Then the system COE composite compliance level, C_s, is computed as follows:

If $C_{ma} < C_{dii}$, then $C_s = TRUNC[(C_{ma} + C_{dii})/2]$,
else $C_s = ROUND[(C_{ma} + C_{dii})/2]$

where **ROUND** means to round the result to the nearest integer.

As an example, assume that a system has five mission applications with compliance levels of 5, 7, 7, 8, and 8. Assume that the DII compliance level for the COE segments actually used in the system is 6. Then the system composite compliance level is

$C_{ma} = 5$
$C_{dii} = 6$
$C_s = TRUNC[(5 + 6)/2] = 5.$

If the least compliant segment (level 5) could be improved to reach level 7 with no change in the COE, then the resulting system compliance level would be increased to 7.

2.1.4.4 Compliance in Mixed Systems

COE-based systems are likely to be created which include a mixture of COE-based and non-COE based computing platforms. This may occur for several reasons:

1. because required functions in the target system have not yet migrated to the COE,
2. because of the need to interface with legacy systems that are not COE-based (e.g., mainframe applications),
3. because the COE is not presently available on a required platform, or
4. because the platform is highly specialized and is not appropriate for the COE.

An example of the latter situation is a receiver subsystem that contains dedicated hardware for direct receiver control. A system built around such components is likely to use a platform on which the COE is available for operator interaction and for receiver tasking, and hence would be a mixed system.

Calculating the system composite compliance for all four situations is done just as with COE-based systems described in subsection 2.1.4.3. In the first situation above, the application that contains the required functionality can still be evaluated against the compliance checklist and so arrive at a compliance level. The resulting system compliance level will likely be very low.

Computation of the system composite compliance in the last three situations is equally straightforward. Compliance is computed by ignoring the legacy platforms and platforms for which the COE is not available.

2.1.5 Interoperability of COE-Based Systems

This subsection describes interoperability in the context of the COE, and shows the relationship between DII compliance levels and interoperability. But first, it is important to distinguish between *interfacing*, *integration*, and *interoperability*. The three terms are closely related and often confused, but they are distinct concepts. Proper understanding of the interrelationship of these three terms makes it clear that the DII COE is an approach towards integration that goes beyond simple interfacing or “peaceful coexistence” to true interoperability.

2.1.5.1 Interfacing Systems

Interfacing is the ability of two systems to exchange data, typically by converting data to an agreed-upon intermediate format. Interfacing should be viewed as one approach towards achieving interoperability, or as a first-level approximation of interoperability. For example, military systems frequently interface with one another by exchange of United States Message Text Format (USMTF) messages. They are able to "interoperate" to the extent that they can pass meaningful data to one another in an agreed-upon intermediate USMTF format.

Interfacing provides a limited degree of interoperability but fails to fully satisfy "real-world" operational requirements. Interfacing

- requires consistent interpretation of the agreed-upon format for data exchange;
- requires systems to stay in synch as the data exchange format changes;
- may result in loss of precision or other attributes (e.g., one system may process latitude and longitude only in degrees and minutes, while another system may process latitude and longitude down to decimal fractions of a second); and
- fails to ensure that applications interpret the exchanged data consistently.

For these reasons, successful "interfacing" is often limited to a specific version of the two systems in question and may not survive when an upgrade to either system is performed. Also note that standards profiles specify how interfacing can be accomplished.

2.1.5.2 System and Segment Integration

Integration is often used to refer to integration **within** a system or **between** systems, or to refer to software and data integration. Within the context of this document, integration refers to combining segments to create a system. Segment integration refers to the process of ensuring that segments:

- work correctly within the COE runtime environment;
- do not adversely affect one another;
- conform to the standards and specifications described in this document;
- have been validated by the COE tools; and
- can be installed on top of the COE by the COE installation tools.

Integration requires resolution of compatibility issues between components that are to be interconnected. Integration attempts to allow sharing of a common resource (such as data) without the need for intermediate translations from one format to another. Note that the COE is a technique for achieving both software and data integration; it is the SHADE component of the COE which the technique for assuring data integration. But the DII COE goes further because COE/SHADE-type integration for software and data provides true interoperability as a byproduct. The COE with full SHADE does not create any technical roadblocks to interfacing, but does strongly encourage a deeper level of integration that promotes true interoperability.

Integration of a segment with the COE is the responsibility of the segment developer. Government integrators perform integration of the system as a whole and interoperability testing.

2.1.5.3 Interoperability Levels

In the context of this document, *interoperability* refers to the ability of two systems to exchange data:

- with no loss of precision or other attributes,
- in an unambiguous manner,
- in a format understood by and native to both systems, and

- in such a way that interpretation of the data is precisely the same.

There are two significant differences between interoperability and interfacing. The first is that with interoperability the exchange of data is performed without the need to translate to an intermediate format, such as a USMTF message format. This leads to the second difference in that interoperable systems will produce exactly the same "answer" in the presence of identical data. Systems that are interfaced will not necessarily do so because of the potential loss of precision or data in the data exchange.

The concept of interoperability is explored in more detail in a study sponsored by the C4I Surveillance and Reconnaissance (C4ISR) Integration Task Force Integrated Architectures Panel. The draft document proposes four levels[21] of interoperability which are adopted by the *I&RTS*. The four proposed levels are as follows, listed in decreasing order of interoperability:

> **Level A: Universal - Virtual C4I System.** This level represents the ultimate goal of full interoperability. Universal interoperability is characterized by the ability to globally share integrated information in a distributed information space. Another way to view Universal interoperability is as a way to globally share systems.
>
> **Level B: Advanced - Integrated Systems.** The Advanced level of interoperability is characterized by shared data between applications, including shared data displays, and information exchange through a common data model. This level provides for sharing of information in a distributed but localized environment and for sharing of applications.
>
> **Level C: Intermediate - Distributed Systems.** This level is characterized by a client/server environment with standardized interfaces and distributed computing services that allow for exchange of heterogeneous data (e.g., maps with overlays, annotated images), and advanced collaboration. This level of interoperability is achievable with implementation of "cut and paste" between applications, through World-Wide-Web technology, and through basic use of DII COE features.
>
> **Level D: Basic - Discrete Systems Interaction.** A primitive level of interoperability characterized by peer-to-peer connected systems that allows basic exchange of homogenous data (e.g., email, formatted messages) and allows for basic collaboration. This level of interoperability is achievable by interfacing techniques described above and by use of standard office automation products that provide data import/export functions for handling data from another product.

2.1.5.4 Mapping Interoperability and Compliance Levels

Note that progressing from one level of interoperability to a higher one requires a deeper degree of integration, more commonality in the infrastructure building blocks, and a greater ability to share data and information. These are precisely the requirements for progressing to deeper levels of DII compliance, and can be achieved through the use of COE/SHADE facilities. When two operators are using exactly the same system, or two systems which are nearly identical, they achieve the highest possible degree of interoperability. The more software reuse is achieved, the greater the degree of interoperability. Thus, there is a direct relationship between integration, reuse, DII compliance levels, and interoperability.

Integration alone does not imply interoperability; it only provides a level of assurance that the system will work as designed. However, when COE-based systems are integrated together, interoperability is achieved as a byproduct because common software is used for common functions. The degree to which interoperability is achievable is dependent upon the degree to which the two systems are DII-compliant. Universal Interoperability can only be achieved when systems use exactly the same software to perform

[21] The draft document also proposes a mapping between the *I&RTS* compliance levels and interoperability levels. However, the mapping fails to properly account for integration when *identical* software is used for common functions.

identical functions and use the same database segments for required data elements. Implementation of agreed-upon paper standards is not itself sufficient.

Table 2-1 shows a mapping between DII compliance levels and interoperability levels. The transition and correspondence between levels is not sharp, as the table suggests because the purpose and focus are different for the two different types of levels.

DII Compliance Levels	Interoperability Levels
1. Standards	Basic
2. Network	Basic
3. Platform	Basic, Intermediate
4. Bootstrap	Basic, Intermediate
5. Minimal	Basic, Intermediate
6. Intermediate	Intermediate
7. Interoperable	Intermediate, Advanced, Universal
8. Full	Advanced, Universal

Table 2-1: Compliance and Interoperability Levels

2.1.6 Principles for Selecting COE Components

Selection of the specific software modules that comprise the COE determine which mission domain(s) can be addressed by a particular COE reference implementation. But selection of COE components is not arbitrary: it is driven by a number of important architectural and programmatic principles. First, there is a determination of what functions the COE is required to perform, then there is a set of criteria for selecting software components which perform the required functions. A function is part of the COE if it meets one or more of the following criteria:

1. *The function is part of the minimum software required to establish an operating environment context.* This is normally provided by COTS products and includes the operating system, windowing software, security software, and networking software.

2. *The function is required to establish basic data flow through the system.* To be useful, a system must have a means for communicating with the external world. To be efficient, consistent, and robust, a system must also have standard techniques for managing data flow internal to the system.

3. *The function is required to ensure interoperability.* Standards alone cannot guarantee interoperability, but using common software for common functions and using shared and universal database segments with DOD 8320 standard data objects comes much closer. As an example from the GCCS mission domain, a USMTF message parser is part of the COE because interoperability cannot be achieved if two different message parsers implement a different set of assumptions about the USMTF message specification or use a different specification revision.

4. *The function is of such general utility that if rewritten it constitutes appreciable duplicative effort.* This includes printer services, an alerts service for disseminating alerts, and a desktop environment for launching operator-initiated processes.

Subsections 2.1.1 and 2.1.2 detail the functions currently defined to be in the DII COE. The first three criteria listed above are technical in nature because they dictate from an architectural perspective what software *must* be contained in the COE for a given mission domain. The fourth criteria, however, is more programmatic in nature because it is often a tradeoff between the cost of modifying a legacy system to remove duplication versus the cost of maintaining duplicative code, the cost of potentially requiring additional hardware resources because of duplication, and the cost of operator training when there are

different ways to accomplish the same action. DII compliance requires that there be no duplication of functions in the first three criteria but some flexibility is possible for the fourth.

There are two frequently voiced concerns about COE services:

1. if a module becomes part of the COE, it cannot be easily changed or customized; and
2. the larger the COE is, the more inflexible and the poorer the performance of the resulting system.

The first statement is partially true and is so by design. It is essential to perform careful configuration management of COE components, and they must be changed only in a controlled way in response to formally reported problems. Stability of the COE is crucial to the system, so modifications must be done carefully, deliberately, and at a slower pace than changes in non-COE routines. But just because changes are controlled does not mean that the COE routines cannot be customized. Ongoing work in the COE is to devise and refine techniques to "open up the architecture" to allow applications to customize COE components in ways that do not violate COE principles and do not adversely impact other developers using COE services.

The second statement is a misunderstanding of the COE architecture and concept. Unlike many systems, the COE is not designed as a single monolithic process, but is instead designed as a collection of relatively small processes. While a small number of these are loaded into memory as background processes, most are loaded into memory on demand in response to operator actions (e.g., edit a file, display a parts inventory) and only for the amount of time required for them to perform their task. This approach offers considerable flexibility because it limits the number of background processes required. Except for cases where segments require adding new background processes, adding new segments does not adversely impact performance. The price paid is a small amount of overhead required to load functions on demand, but this is generally negligible because the overhead is small and comes usually in response to an operator request to bring up a display that must respond only at human speeds.

A COE-component segment is not necessarily installed on every target platform or as part of every COE-based system. A COE-component segment can be omitted from the system or installation if:

- *Any remaining COE-component segments do not require the functionality provided by the segment.* For example, the COE provides a number of message parsers for processing military message formats. But systems such as ECPN have no need to handle military message formats and therefore such parsers need not be included in the ECPN system. However, in many cases there is no real advantage to deleting a COE-component segment because it will not be activated unless required and the amount of disk space taken up is small. Eliminating the function will increase the burden of configuration management problems more than leaving the function in the system.

- *The functionality provided by the segment is not required by any remaining COE-component segments.* Selection of certain functions within the COE automatically dictates the inclusion of segments on which those functions depend. This is not the same as saying that the COE is not modular. On the contrary, it is an observation that inclusion of a higher-level function requires inclusion of all lower-level routines used to build the function. This is a direct consequence of modularity, not a contravention.

- *The functionality provided by the COE segment is not duplicated by another segment.* A common pitfall to avoid is omitting a COE component because its functionality is available through some other means. The problem with this approach is that a common "look and feel" and consistent operation are no longer preserved between applications, and interoperability may be reduced.

Omission of COE-component segments that are not required is done automatically by the COE installation software.

2.2 Account Groups and Profiles

In a typical operating system, users are assigned individual login accounts. Configuration files are created to establish user preferences and a runtime environment context. In the UNIX operating system, configuration files (for example, `.cshrc`) establish the runtime environment context for the user. COTS products such as CDE also have configuration files that contribute to the runtime environment context as well. These configuration files must be set up and established for each user of the system. The COE provides standard versions of the required "dot" files. These should be used when creating account groups because they standardize the operation of the system across all account groups, and because the COE-provided files demonstrate how to support dynamic profile switching.

An *account group segment* is a template used within the COE for setting up individual login accounts and a required runtime environment context. Account groups contain template files for specifying items such as the functions to be made available to operators and global default preferences such as color selections for window borders.[22] Account groups are described further in Chapter 5 of this document.

Account groups can also be used to perform a first-level division of operators according to how they will use the system. This technique is used in the COE to identify at least five distinct account groups:

- Privileged Operator (e.g., root) Accounts,
- System Administrator Accounts,
- Security Administrator Accounts,
- Database Administrator Accounts, and
- Non-Privileged Operator Accounts.

Other account groups may exist for specialized system requirements, such as providing a character-based interface, but all account groups follow the same rules. Within an account group, subsets of the available functionality can be created. These subsets are called *profiles*. An operator may participate in multiple account groups with multiple profiles, and can switch from one profile to another without the need to log out and log in again. An operator may also select multiple actives profiles to provide an operational environment from a collection of account groups. For example, assuming the operator has appropriate permissions, an operator may combine a profile based on the System Administrator account group with a profile based on a Database Administrator account group.

Figure 2-6 shows the hierarchical relationship among account groups, profiles, and individual users. It is intended to convey several points.

- Multiple profiles may be assigned to an account group, but a particular profile may be assigned to only one account group. Assuming the operator has proper permissions, multiple profiles may be selected at one time to give the operator features from multiple account groups at the same time.

- Multiple operators may be assigned to the same profile. For example, operator Op4 and operator Opn are shown assigned to the same profile within the Non-Privileged Operator account group.

- Operators may be assigned to multiple profiles either within the same account group, or across account groups. Opn is assigned to three profiles within the Non-Privileged Operator account group. Op3 is assigned to a profile in the System Admin, Security Admin, and Database Admin account groups.

- Not only can an operator be assigned to multiple profiles, but multiple profiles may be active at a time. The operator may switch between profiles without the need to log in and out. (Optionally, a system can be configured to permit a single profile at one time.)

[22] The user may modify preferences, but the Account Group establishes the initial, default settings.

- The COE allows profiles to be locked. That is, if Op4 and Opn are assigned to the same set of profiles, the system can be configured so that if Op4 is in a specific profile first, then Opn is locked out from using that profile until Op4 is no longer using it.

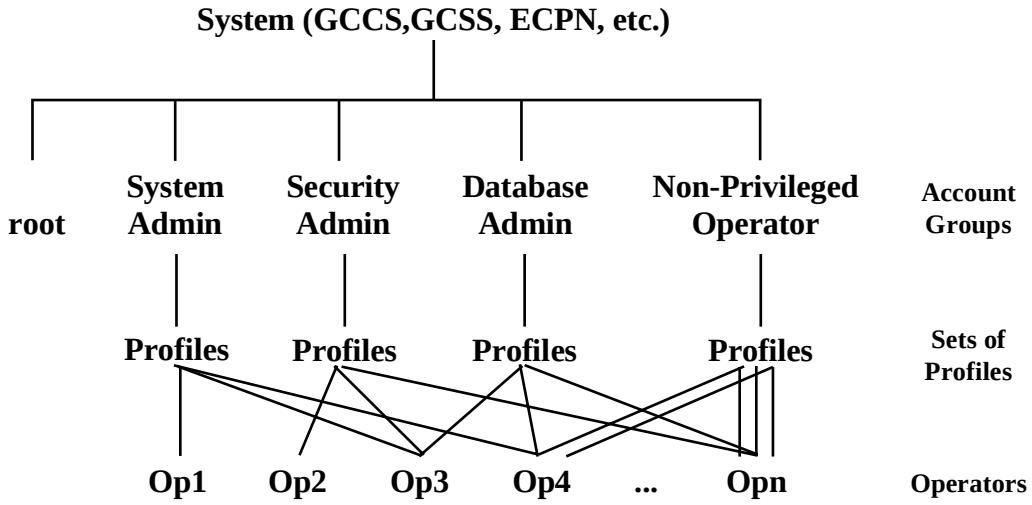


Figure 2-6: Account Groups, Profiles, and Users

2.2.1 Privileged User Accounts

Most operating systems provide a privileged "super user" account. Both UNIX and Windows NT have the concept of a privileged account. Privileged accounts are normally restricted to knowledgeable systems administrators because serious damage can be done to the system if they are used improperly. Security requirements also dictate careful control and auditing of actions performed when operating as a privileged user.

The COE design philosophy is to not require the use of a privileged user account for normal operator activities. Certain processes cannot be performed without superuser privileges, but such privileges should be given to the process, not the user, and only for the period of time necessary to perform the required action. Root-level access need *not* be provided to the user for such actions: indeed, it should not be provided.

Normal operation does not require a command-line-level access, especially to root. Command-line access for any COE segment is expressly prohibited unless the DISA DII COE Chief Engineer grants prior approval. However, a privileged user account is preserved in the system for use by trusted processes, for unusual system administration tasks or installations, and for abnormal situations where "all else fails."

2.2.2 Security Administrator Accounts

Security in the COE is implemented through a collection of security services and trusted applications. One such trusted application is the Security Administrator application that allows a Security Administrator to monitor and manage security. Precise functionality of the Security Services provided is vendor-dependent because vendors have taken different approaches. Security features in Windows NT and UNIX are significantly different, but even within UNIX, security features vary considerably from one vendor to another.

The Security Services are loaded as part of the COE kernel. (The precise sequence for loading security software is vendor-dependent.) The Security Administrator application is designed to be made available to only a restricted group of operators. Available functions include the following:

- Ability to create individual login accounts
- Ability to create defined operator profiles, including granting database privileges as established by the Database Administrator (DBA)
- Ability to create/modify database user accounts
- Ability to assign read, write, and modify data permissions
- Ability to customize menus by operator profile.

2.2.3 System Administrator Accounts

The System Administrator Account Group is a specialized collection of functions that allow an operator to perform routine maintenance operations. This software is designed to be made available to a restricted group of operators. It is loaded as part of the COE kernel because it contains the software required to load segments. Functionality provided includes:

- Ability to format floppy disks
- Ability to install and to remove segments
- Ability to set platform name and IP address
- Ability to install and configure printers
- Ability to create and to restore backup tapes
- Ability to shutdown and to reboot the system
- Ability to configure network host tables
- Ability to configure and manage the network.

2.2.4 Database Administrator Accounts

The Database Administrator Account Group is to be used by those individuals responsible for performing routine database maintenance activities such as backups, archives, and reloads. The specific capabilities are dependent upon which commercial relational database software is in use and upon tools provided with these commercial products.

Functions included within this account group are:

- Ability to archive and restore database tables
- Ability to import and export database entries
- Ability to create/modify database user accounts
- Ability to checkpoint and journal database transactions.

> **Note:** User account management is normally done as part of a Security Administrator account. However, the COE provides the ability to modify the database portion of already created user accounts via the Database Administrator accounts as well. Only those user account items related to database administration can be modified by the database administrator.

2.2.5 Operator Accounts

Most operators will not require, nor will site administrators grant access to, capabilities described in the previous subsections. Most system users will be performing mission-specific tasks such as creating and

disseminating Air Tasking Orders (ATOs), preparing briefing slides, performing ad hoc queries of the database, participating in collaborative planning, etc. The precise features available depend upon which mission-application segments have been loaded and the profile assigned to the operator.

2.2.6 Character-Based Interface Accounts

Certain legacy systems require the ability to provide a character-based interface to the user. This is typically required for remote users where the communications bandwidth is too low to support a GUI-based application or because the user's hardware does not support graphics (e.g., VT100 terminals).

The COE provides a character-based account group for such situations. These may have profiles defined just as with any other account group. When the user logs in, a menu of options, such as

0) Exit
1) AdHoc Query
2) TPFDD Edit

Enter Option:

is presented to the user.

Character-based account groups are restricted in the sense that a user account is either character-based or it is not. If an operator has access to a terminal that supports a GUI interface and to another that does not, the operator must have two separate login accounts: one which uses character-based profiles and one which does not.

2.3 Site Configurations

Figure 2-7 is a simplified notional LAN diagram for a typical COE-based system. The architecture consists of a 3-tier client/server environment incorporating data servers, application servers, and platforms interconnected on a LAN/WAN. The division shown separates data (data servers), functions (application servers), and presentation (platform). System components are interconnected on a LAN/WAN through direct connection to a LAN, through subnets connected over routers, through dedicated lines, or via dial-up through a communications server. Cryptologic equipment may be installed to secure communications over non-secure lines as shown. Remotes with limited bandwidth will not generally have access to the complete suite of mission applications available to local users.

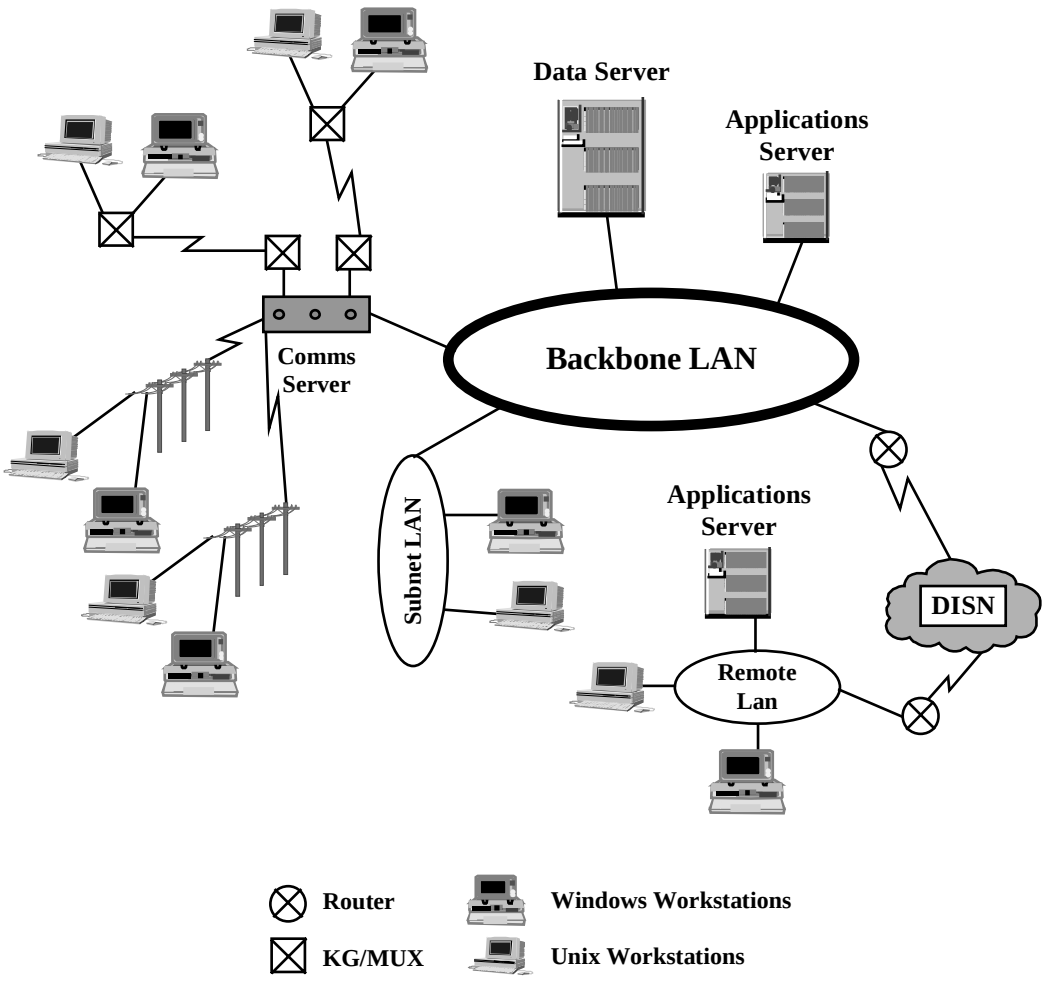


Figure 2-7: DII Notional LAN Architecture

In a typical installation, there will be one or more database servers and several application servers. The database server is the repository for all databases and may be replicated at strategic places in the LAN architecture to improve performance and to balance loading. COE services ensure that replicated databases stay synchronized.

A typical installation will often include other servers that are not shown in the diagram. A Web server connected to the outside world through a firewall allows sites to take advantage of Web technologies for

collaborative planning purposes. As described in a later chapter, the COE also provides facilities that allow developers to create applications that use the Web for accessing applications and data.

Network management is greatly simplified if a domain name server is created and if there is a server for management of user accounts and profiles. Segment installation is simplified by designating servers to load platforms across the LAN rather than individually from magnetic media. This approach also simplifies software distribution because when software updates are received, they can be tested in isolation, then loaded onto a segment server for distribution to affected platforms. Combined with configuration definitions, the COE provides powerful tools for managing software installation and distribution.

The COE supports both “network-centric” and “platform-centric” LAN management. Network-centric refers to the ability to centrally manage network resources (e.g., user accounts, profiles, software installation). In keeping with COE principles, centralized management can be done from any platform (subject to security considerations) with infrastructure services responsible for effecting the changes across the network. Platform-centric refers to the ability to distribute network management. The choice of centralized versus distributed is a preference which may vary across distributions or sites.

2.4 Installing COE-Based Systems

Figure 2-8 is a notional depiction of the installation process. (It should not be interpreted too literally since vendor-specific loading instructions may require slight alterations in the loading sequence shown.) First, the operating system, windowing environment, and any necessary patches are loaded as per vendor instructions. Then, the COE Security Administration software and System Administration software are copied onto disk with the equivalent of a UNIX "tar" command. The segment installation tool is copied onto disk as part of installing the System Administration software and installation of the System Administration software is done in such a way as to also create a System Administrator operator account. This completes installation of the COE kernel. Next, the operator logs in as a system administrator, invokes the segment installation tool, and selects the remaining COE segments for installation. Finally, any remaining mission-specific segments are selected and loaded.

This installation approach has several advantages. It greatly simplifies the installation process by handling all vendor-unique issues first (e.g., loading the operating system and patches). It guarantees a standard, known starting configuration for all platforms regardless of how they will be used. It allows all remaining segments to be loaded in a standard way regardless of the hardware platform or mission application, thus simplifying system administration. Through the COE, it allows segments to extend the base environment as required as they are loaded.

Figure 2-8 describes the general flow for installing a system, which can be accomplished in either a "pull" or a "push"[23] mode. In pull mode, installation is done locally from the target platform. In push mode, installation is performed remotely onto the target platform from a different platform.

Installation may be accomplished in several ways:

- directly from distribution media (e.g., tape, CDROM),
- locally from distribution media mounted on a different platform,
- across the network from a segment server, or
- through a Web browser interface from a centralized segment server.

The distribution media and servers may contain executables for multiple hardware platforms. The segment installation tool ensures that only those executables which are compatible with the target platform (e.g., NT, Solaris, Digital Equipment Corporation [DEC]) are selectable and hence installable.

[23] Installation in a "push" mode requires that the COE kernel already be installed on the target machine and that the target machine already be accessible from the network.

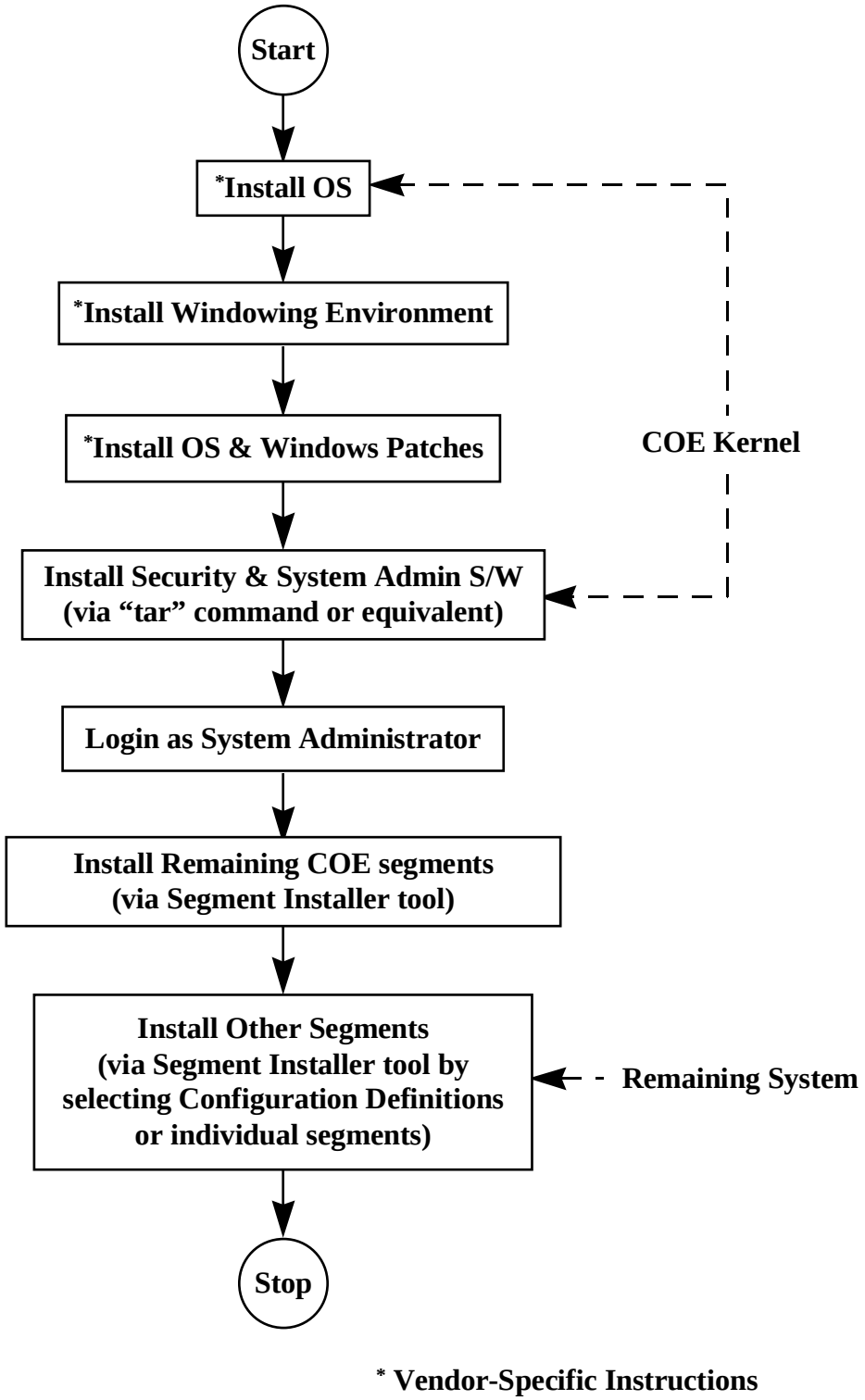


Figure 2-8: Installing a COE-Based System

2.5 Supported Configurations

The DII COE is an open architecture and as such is not tied to a specific hardware platform. It uses POSIX-compliant operating systems and industry standards such as X Windows, Motif, and NT. In actual practice, POSIX compliance and industry standards have not progressed to the point where verification that software works in one hardware/software configuration is a guarantee that it will work in another. COTS vendors do not necessarily provide backwards compatibility with subsequent releases, and in fact much of the effort consumed in porting the COE from one configuration to another is to account for lack of compatibility between vendors or between vendor releases. Thus, what hardware/software configurations to support is more an issue of testing and life-cycle maintenance than it is one of "openness" or software portability.

COE reference implementations exist for a number of platforms. The list of supported hardware and software components is growing as the COE and COE-based systems evolve to meet operational requirements. Appendix A lists the current DISA-supported COE configurations. It also describes a DISA "self-certification" program that allows vendors or services to receive copies of the COE kernel in order to port it to platforms or operating systems not currently supported by DISA. DISA will test the ported kernel to ensure it meets COE requirements and will issue a certification for the specific platform. Responsibility for supporting the ported COE on the new platform is the responsibility of the vendor/service that has funded the effort.

Appendix A will be updated as required to reflect new hardware/software configurations. Note that not all of the COTS products listed in Appendix A are part of the COE kernel and thus are not required for every platform. Refer to the DISA DII COE Chief Engineer for requirements for other platform or COTS software versions, or for an updated list of supported vendor products.

Specifying precise hardware requirements in terms of memory, disk space, etc. is a function of whether the platform is a shared data server, an application server, or a client platform, and whether the platform is standalone or on a network with other COE-based platforms. Consult the DISA DII COE Chief Engineer for hardware configuration options.

3. Development Process Overview

This chapter describes the development process in more detail. A powerful feature of the overall development process is the concept of "automated integration." Automated integration means that automated tools are used to combine and load segments, make environmental modifications requested by segments, make newly loaded segments available to authorized users, and identify places where segments conflict with each other. Traditional system integration then becomes primarily a task of loading and testing segments, although traditional functional testing must still be performed to ensure interoperability and performance of the resulting system. Automated integration has the advantage that traditional integration tasks are pushed as far down to the developer level as possible, and then validated as system integration is performed.

Prior to submitting a component to DISA, a developer must

- package the component as a segment,
- demonstrate DII compliance through tools and checklists,
- test the segment in isolation with the COE,
- provide required segment documentation, and
- demonstrate the segment operating within the COE.

The DISA DII COE Software Support Activity (SSA) enters the segment into the online library for configuration management purposes and confirms DII compliance by running the same suite of tools as the developer. The SSA then tests interaction between segments and the impact on performance, memory

utilization, etc. Since segments typically can only interact through the COE, the task is greatly simplified and the need for human intervention in the process is minimized.

An automated integration approach is a practical necessity. Not only do different services and agencies contribute segments, but individual segments are created by a large body of different developers. Traditional integration approaches rapidly break down with the need to communicate to such a large number of people while the costs incurred to resolve inter-module conflicts at system integration time become prohibitive.

This chapter begins with a consistent approach to version numbering, followed by a detailed look at the development phases. The chapter ends with some special considerations for how to migrate legacy systems rather than developing from scratch. Because of the special importance of the online library, Chapter 10 is devoted to it and its features. For the present chapter, it is sufficient to note that there is a configuration management repository for segments.

> **Note:** Integration and testing of a segment within the COE, and DII compliance are the responsibility of the segment developer. Government directed integrators verify DII compliance, integrate the system as a whole, and perform interoperability testing.

3.1 Version Numbering

The COE concept requires the ability for segments to state dependencies upon or conflicts with other segments. At least four types of segment dependencies/conflicts can exist.

1. One segment may require that another segment also be loaded in order to operate.
2. One segment may require another, but the dependency is version-specific.
3. One segment may have a conflict with another segment so that both cannot be present in the system at the same time.
4. One segment may have a conflict with another, but the conflict may be version-specific.

A consistent approach to version numbering is thus a mandated feature of the COE standard so that the COE tools can detect and enforce segment dependencies, and can detect and avoid segment conflicts. Version numbers are applied to all segments and all segment patches.

COE-based systems consist of a collection of segments, each with its own individual version number. When a version number is applied to a COE-based system, the version number refers to the entire system as a whole, not the version for each individual mission application or segment, or for the COE version. While this may seem confusing at first, it is a practical necessity and is consistent with commercial practice. For example, one refers to the version of Microsoft Windows (analogous to the DII COE) as well as individual applications like Word or Excel (analogous to mission applications like GCCS Status of Resources and Training System [GSORTS] or to COTS products like Netscape). Microsoft packages several of their office automation products into Microsoft Office (analogous to GCCS) and gives the collection a version number, even though it is composed of individual products, each having its own version number. The Microsoft Office package is advertised as requiring a specific version of Windows to operate.

DII compliance mandates adherence to the version numbering scheme outlined in this section. Version numbers are frequently tied to the signature level required to authorize a product release. Hence they have programmatic importance as well as technical importance for distinguishing between segment upgrades.

3.1.1 Segment Version Numbers

Segment version numbers consist of a sequence of 4 integers, separated by decimal points, in the form

a.b.c.d

where each of the integers has a specific meaning. The first integer is a *major release* number and indicates a significant change in the architecture or operation of the segment. Compatibility libraries will be provided if necessary to preserve backwards compatibility. The second integer indicates a *minor release* in which new features are added to the segment, but the fundamental segment architecture remains unchanged. A minor release may necessitate relinking to take advantage of updated API libraries, but APIs are preserved at the source code level except possibly on a documented basis with the explicit approval of the DISA CCB. The third integer is a *maintenance release* number. New features may be added to the segment, but the emphasis is on optimizations, feature enhancements, or modifications to improve stability and usability. APIs are preserved and do not generally require segments to recompile or relink during successive releases. The fourth integer is a *developer release* number.

For COE segments, the first three integers are assigned by DISA. For mission-application segments in a COE-based system such as GCCS, the program manager assigns the first three integers. In both cases, the final integer is reserved for developers. The fourth integer is updated to keep track of successive releases during the integration process.

Version number integers are always incremented, never decremented, to indicate later releases of a segment. This scheme provides a readily apparent method of comparing successive releases of a segment. For example, a segment with version number 2.1.6.1 is a newer version than 2.1.0.5. Moreover, according to the scheme outlined, APIs are preserved. Segments using version 2.1.0.5 can usually be expected to work without any modification when loaded on a system using the 2.1.6.1 version.

When specifying version dependencies, this scheme also allows segments to indicate the degree to which they are version sensitive. For example, suppose Segment A requires use of Segment B. Segment A may indicate that it requires Segment B, version 2.3 indicating that any maintenance release of version 2.3 (e.g., 2.3.2.0, 2.3.1.2) is acceptable. The same approach works for specifying segment conflicts.

> **Note:** It is a violation of the COE to fail to increment version numbers between subsequent segment releases. This applies to *all* segments whether they are COTS segments, COE-component segments, or mission-application segments. This requirement to update version numbers between subsequent releases is a matter of good Configuration Management practice.

3.1.2 COTS Version Numbers

COTS products will typically already have version numbers assigned to them, but the convention used is vendor-specific. This makes it difficult to make meaningful version comparisons in the same sense as in the previous subsection. A further complication is that COTS products must often be configured before they can be properly utilized in a COE-based system. For this reason, COTS segments are also assigned version numbers.

A COTS version number consists of a *primary* and *secondary* version number separated by the '/' character. The primary version number follows the same convention described in the previous subsection, while the secondary version number is the version number assigned by the vendor and can be any alphanumeric string. Comparisons and dependency specifications are always performed using *only* the primary version number. Secondary version numbers should be specified because they may be used for other purposes such as supporting automated license management.

For example, the DII COE requires an increase in the amount of shared memory configured in the vendor-supplied Solaris 2.5 UNIX Operating System. A primary version number, such as 2.1.3.6, is assigned so

that the operating system is referred to as version 2.1.3.6/SOL-2.5. Similarly, the X11R5 version of an X Windows server might have a version number assigned such as 2.3.0.4/X11R5.

COE-based systems are presently composed of segments contributed from ongoing programs that may already have an established convention for version numbering. A secondary version number may also be attached to such segments. As with COTS segments, only the primary version number is actually used within the COE.

3.1.3 Patch Version Numbers

Patches[24] are indicated by appending the letter 'P' and a single number to the primary version number. For example, patch 12 to version 2.1.3.5 of a segment would be designated as version 2.1.3.5P12. Patch 4 to the Solaris Operating System example in the previous subsection would be designated as 2.1.3.6P4/SOL-2.5.

3.1.4 COE Version Number

The DII COE itself is composed of a collection of segments. Each of these has its own version number, but it is convenient to track the COE as a single entity. For this reason, DISA assigns a single version number[25] to refer to a specific release of the collection of segments in the COE. Mission applications may thus state dependencies on the COE as a whole rather than individual segments within the COE.

It is possible that some mission applications need to state a dependency on a particular segment within the COE. This should normally not be required, but is permitted.

3.1.5 System Version Number

A COE-based system is comprised of COE segments, and mission-application segments. Each of these segments will have their own individual version, but it is usually more convenient to the end user to view the system as a whole rather than as a collection of individual pieces. Thus, it is advisable to assign a single version number to the whole to refer to the system rather than confusing the end user with a list of segments and their associated version numbers. Identification of the system version number is the responsibility of the cognizant DOD program manager. It is also the responsibility of the cognizant DOD program manager to track the track the segment versions that are to be associated with a particular system release. The version number should be that identified in the main operator account group (e.g., GCCS, ECPN, GCSS).

For example, suppose that the system GCCS, version 3.2, is to be comprised of the following segments or groups of segments:

- DII COE, version 4.0
- JOPES, version 3.2.1
- GSORTS, version 5.6.3.2
- Scheduling and Movement (S&M), version 1.0.3

Then the cognizant DOD program manager should

1. enter "3.2.0.0" as the version number in the GCCS account group (see Chapter 5 for more information on account groups and how to enter a version number); and

[24] A patch in this context is the total replacement of one or more files, not the replacement of a subset of a file or a section of memory. The files being replaced may be software or data.

[25] The version number of the COE must not be confused with the version number for the *I&RTS* document. The two are not related. One is the version number of a delivered software product while the other is the version number of a specification document.

2. in accordance with good Configuration Management practices, maintain a list of the exact segments and versions that comprise this GCCS system release.

Note: The COE provides an environment variable, `COE_SYS_NAME`, that the account group must set to provide the system name. See Chapter 5 for more details.

3.1.6 Configuration Definition Version Numbers

As described in Chapter 2, the COE provides configuration definitions to simplify management and installation of COE-based systems. Version numbers should also be assigned to properly track changes to configuration definitions. Refer to the appropriate programmer's guide for details on assigning version numbers to configuration definitions.

3.2 Process Flowchart

Figure 3-1 is a representative flowchart of the development process, beginning with registering a segment to be developed and ending with ultimately installing the segment at an operational site. The major development phases are delineated by dashed lines in the figure and correspond to the subsections that follow. This process flow is the same for all segments, including patch segments. As can be seen, the process is indeed largely automated.

By necessity, the figure is abbreviated and does not show several key elements of the development process such as error tracking and reporting, a configuration control board, DISA architecture groups, or configuration management and quality assurance. Each of the elements is strongly implied by Figure 3-1, but their description is beyond the scope of this document. Contact the DII COE Chief Engineer for more information on related elements in the development process.

At several places in Figure 3-1, segments are added to the online library. Segments are compressed and encrypted to reduce disk space and for added security. Segments are also encrypted and compressed when they are transmitted electronically across the network. These actions are performed automatically and are transparent to the user.

While electronic transmission of segments across the network is the preferred approach, it is not possible in certain cases. It is not practical to transmit the operating system, X Windows, or Motif across the network due to licensing restrictions and their size. Other segments, especially the data segments providing fill for database segments, may be too large to send electronically or may have a security classification that requires special handling and tracking. Figure 3-1 should be understood with this in mind. Electronic transfer is performed when feasible, but an alternate route using tape or other media is used as well when required.

Figure 3-1 also shows several places, especially in the Segment Submission phase, where a “Notify” action occurs. This is an electronic notification of status to the segment developer, to the development community, or to the user community. The subsections below describe notifications in more detail, but obviously notifications of status are sent only to the cognizant parties, not necessarily to the entire community. Notification is accomplished by email, WWW, newsgroups, or “paper” as appropriate.

The very nature of COE-based systems dictates that security measures be taken to prevent unauthorized disclosure or access to sensitive information, including project status or system problem reports. For this reason, access to software and project information is divided between Internet and SIPRNET with firewalls to restrict access. This level of detail is not necessary for the overview presented in this chapter and has been omitted from Figure 3-1.

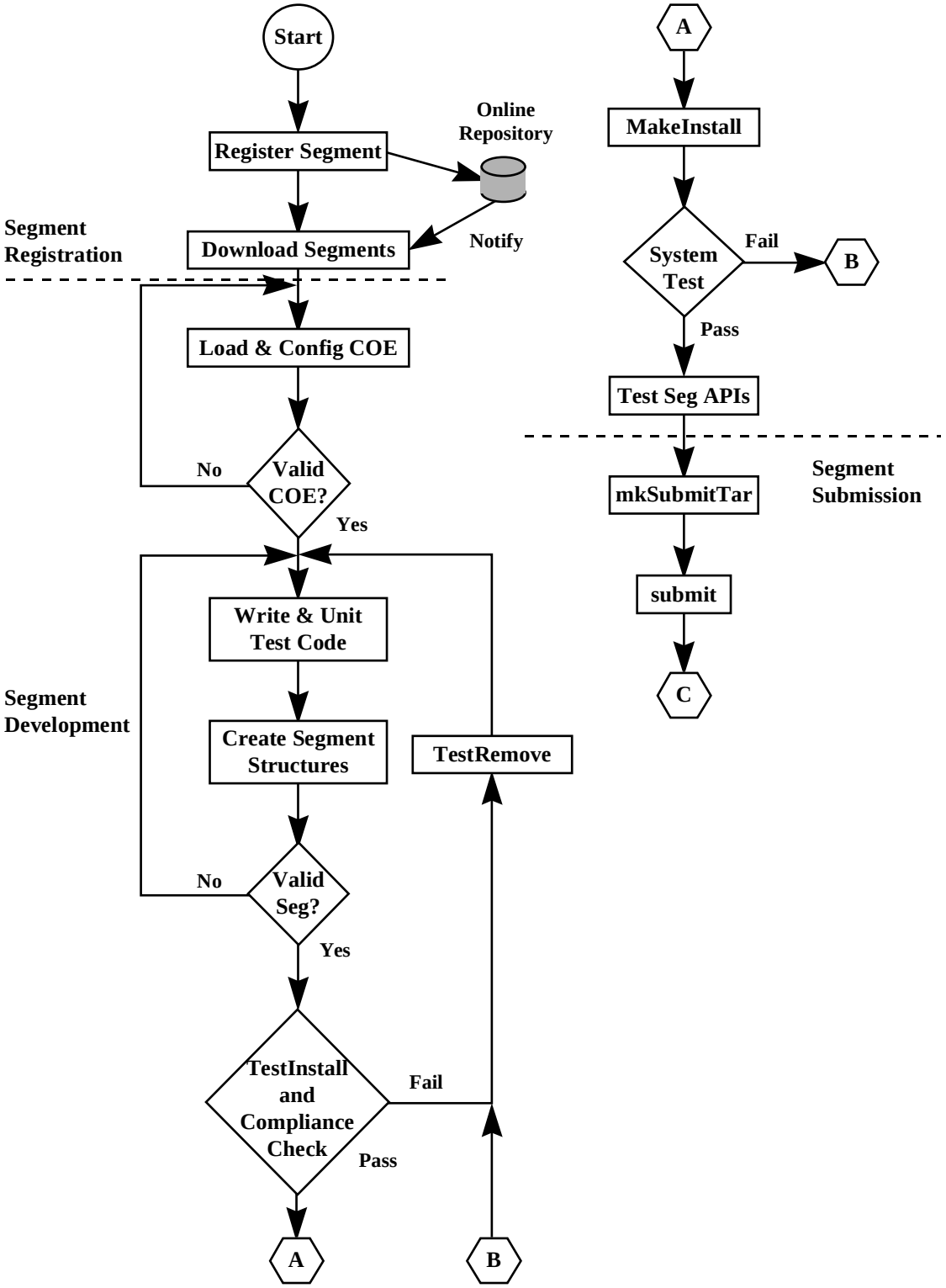


Figure 3-1: Development Process Overview

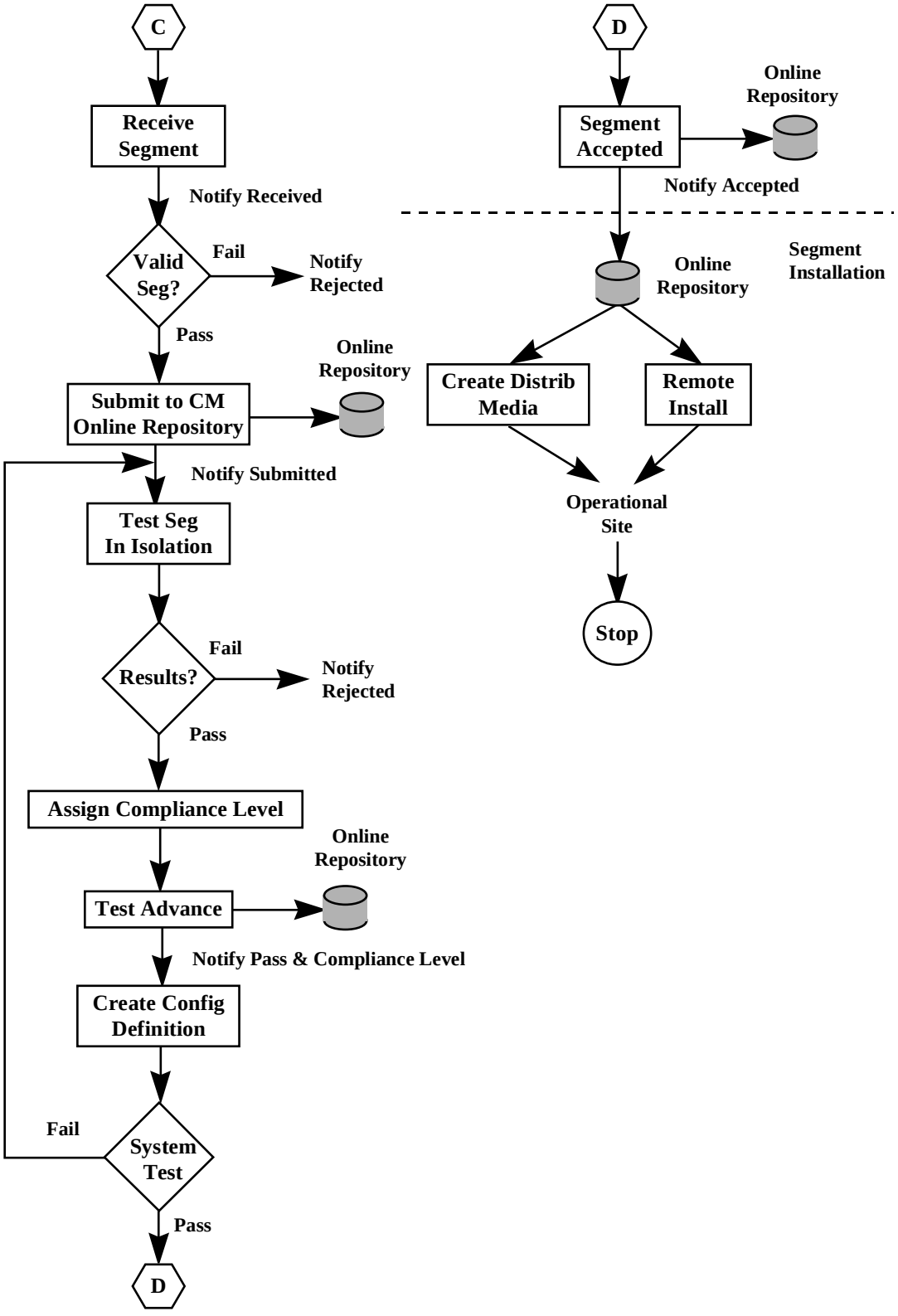


Figure 3-1: Development Process Overview (cont.)

The remaining subsections discuss the process overview in detail. The flow is the same for software segments and for database segments, but there are some additional nuances for database segments. Thus, subsection 3.2.1 describes the process regardless of segment type, while subsection 3.2.2 has additional information for database segments.

3.2.1 Processes Applicable to All Segments

Segment development is straightforward. It essentially requires registering the segment to make sure it will not conflict with other segment developers, create the code, structure the product as a segment, and then test it.

The process in Figure 3-1 is a generic flowchart for any type of segment. The *I&RTS* requires that all COE-component segments be submitted to the DISA DII COE SSA for test and configuration management. Mission-application segments must be submitted to the SSA identified by the cognizant DOD program manager. DISA[26] requires all mission-application segments for DISA Joint Interoperability and Engineering Organization (JIEO) systems (e.g., GCCS, GCSS, ECPN) to be submitted to the same SSA as for COE-component segments.

3.2.1.1 Segment Registration

Segment Registration is the entry point into the development process. Its purpose is to collect information about the segment for publication in a *segment catalog.* Perhaps the most difficult part of maintaining a software repository is simply knowing what capabilities exist. This is the purpose of maintaining a DII segment catalog. The segment catalog is available online through a HyperText Markup Language (HTML) browser and contains information provided by developers in a segment registration form. Keyword searches can be performed on the catalog by developers to identify reusable segments or by operational sites to find new mission applications.

The segment registration form includes, but is not limited to, the following information:

- segment name
- segment prefix
- segment directory name
- segment type (software, data, COE component, etc.)
- system resources (e.g., port assignments, UIDs requested, RPC addresses requested)
- estimated memory required by the segment
- estimated disk storage requirements
- list of boot and background processes (see Chapter 5)
- releasability restrictions (especially export restrictions)
- platform availability (PC only, Solaris only, etc.)
- short paragraph describing the segment features
- unclassified picture of the segment's user interface (GIF, JPEG, or X11 Bitmap format).
- authorization keys (assigned by DISA)
- list of related segments
- list of keywords for use in catalog searches
- program management point of contact

[26] Program managers who do not elect to use the DISA DII COE SSA for their mission applications must coordinate with DISA to ensure that there are no conflicts between their mission-application segments and COE-component segments. If the DII COE SSA is not used, it is the program manager's responsibility to ensure that there are no conflicts with mission applications from other program managers. Since DISA uses a centralized SSA for all DISA JIEO systems, the DII COE SSA manages conflicts between programs (e.g., ECPN, GCSS, GCCS).

- technical point of contact
- process point of contact

The *segment name* can be any character string that is unique among all segments. Segment names for COTS products should usually not include the vendor's name since this will make any segments that depend upon the product vendor-specific. That is, a segment name such as

Company A DCE

is inappropriate because segments that are dependent upon DCE will have to have their dependencies changed if a different vendor is chosen to supply DCE. Refer to Chapter 5 for specific rules regarding selection of a segment name.

Each segment is assigned an identifier called a *segment prefix*. The segment prefix is a 1-6 alphanumeric character string that is used to prevent naming conflicts between segments. Use of the segment prefix is required in any situation where there is the *possibility* that two different segment developers might choose the same name for a public symbol such as an environment variable, executable, API, or library. Two segments may in fact have the same segment prefix as long as there is *no possibility* that public symbols will conflict. This is realistic only if one developer creates both segments.

Segment directory names are often the same as the segment prefix, but they do not have to be. Segment directory names can be any name that conforms to rules imposed by the target operating system, provided they consist only of printable[27] characters, begin with an alphanumeric character, does not end in a blank, and are not already in use by another segment. It is recommended that directory names be limited to 14 characters to avoid porting problems. Refer to Chapter 5 for a specific discussion of how segment name, segment prefix, and segment directory name are used to uniquely identify a segment.

> **Note:** The COE stipulates the same requirements for choosing directory names and filenames as are stipulated for segment directories, except that uniqueness is required only of the segment directory name.

At segment registration time, system resources must be identified. These include estimates of memory and disk requirements. System resources that must be shared and coordinated among other segments must also be identified. These include shared memory estimates, port assignments (e.g., `/etc/services` entries, reserved UIDs), and any other resources that might cause conflicts between segments.

Some segments need access to certain restricted privileges provided by the COE. For example, some segments need to have root privileges to be properly installed. Also, authorization must be granted by DISA before a COE-component segment can be created. When such specialized requests are received and authorization is given by DISA, the DII COE SSA will give the requesting segment developer one or more authorization keys. Unless these keys are provided with the segment, the COE tools will refuse to honor requests for restricted services.

Not all information provided at segment registration time is made available to the community at large. The *technical point of contact* is available only to the DISA Engineering Office in the event that technical questions or issues arise during segment integration. The *process point of contact* is the individual authorized by the segment program manager to actually submit the segment or to receive status information and notifications. The *program management point of contact* is the only individual authorized to commit schedule or resources and is the only individual authorized to release information about the segment to the community at large. The three points of contact are selected by the service/agency responsible for the

[27] Some operating systems allow "." to separate file extensions from filename. Some allow hyphens and underscores. Thus, the COE requires only that the filename be printable, and not begin or end with a blank character.

segment. Services may elect to designate a single individual for all three points of contact, and may include an alternate point of contact for each category.

Referring to Figure 3-1, two steps constitute the Segment Registration phase:

1. *Register the segment.* The segment registration form can be submitted in written form, through email, or in HTML format. Appendix E contains more information on how to do this. Once the developer submits the registration form, the information is entered into the online repository and confirmation is sent to the process point of contact. Segment information is entered into the segment catalog with a tentative release date for the segment. The segment prefix and directory requested will be granted unless they have already been assigned to another developer's segment.

2. *Download segments required for development.* When notification is received that segment registration was successful, developers may download COE-component segments, developer toolkits, object code libraries, and other segments required for software development. Appendix D provides more information on how to download segments, tools, libraries, etc. It also provides information on how to access and search the online segment catalog.

3.2.1.2 Segment Development

The COE approach is designed to be non-intrusive; it places minimal constraints on how developers build, test, and manage software development. Developers are free to establish a software development environment that is best suited for their project. The COE specifies no particular programming language because it is only concerned with handling the resulting executable, not the process or language used to create it. The COE requires only that deliveries be packaged as segments, that segments be validated before submission, and that segments be tested in the COE prior to submission. Figure 3-1 assumes this degree of freedom and omits steps such as design reviews and code walk-throughs that are an expected part of any development effort.

1. *Load and configure the COE.* Most developers will find that the COE will meet their needs as is. However, for some developers the COE kernel may need to be extended to increase shared memory size, message queue sizes, add sockets, etc. Any changes to the downloaded COE must be carefully recorded as environment extensions. It is the responsibility of the segment to request that the COE installation tools make these extensions as the segment is installed. Doing such extensions other than by using the installation tools is a violation of the COE.

2. *Verify that the COE is valid.* The tool `VerifyCOE` checks the integrity of the COE and should be run any time a modification is made to the COE kernel to ensure that the resulting environment is still COE-compatible. It also checks security-relevant features to be sure they have not been adversely modified.

3. *Write and unit-test code.* Develop and test a baseline version of the new software segment as independently of COE software as is possible, but within an environment as nearly identical to the actual runtime environment as is possible. The purpose of this step is to resolve problems within the segment and identify potential interface problems between the segment and the COE, especially the runtime environment. The simplest approach is to launch the segment executables from a command-line prompt within an xterm window (or equivalent) and look for software bugs or conflicts with the COE. The focus of this step is to verify that the segment is correct *internally*.

4. *Create segment structures.* The focus of this step is to verify that the segment can interface *externally* with the COE. Chapter 5 identifies information required to describe a segment through use of segment descriptors. Decisions should be made at this point whether to package data and software together or as separate segments, how best to include any required environment extensions, how to handle segment installation and removal, which features should be icons versus menu entries, etc.

5. *Validate the segment.* The tool `VerifySeg` must be run against all segments to confirm Runtime Environment (Category 1) DII compliance. `VerifySeg` must be rerun when *any* file within the segment that will be present at runtime is modified. This includes segment descriptor files, datafiles, and executables. A segment cannot proceed any further in the process until `VerifySeg` confirms its validity. COE tools used later in the process will reject a segment that has not passed `VerifySeg`.

6. *Install and test the segment.* The tool `TestInstall` allows a segment that is already present on the disk to be installed exactly as if it had been loaded from distribution media at an operational site. When installed successfully, it should be accessible from any operator login that has a profile that has been set up to include the segment. At this stage, it should not be necessary to launch executables from a command line or by any other interim technique. If the installation and test are not successful, the tool `TestRemove` will undo the side effects of installing the segment, but will not delete the segment from disk.

7. *Evaluate the segment's DII compliance level*. As part of the segment test, the developer must evaluate the segment's compliance[28] using the *I&RTS* and *User Interface Specification* checklists. The checklists are to be submitted with the segment as part of the segment delivery.

8. *Create an installation tape.* The tool `MakeInstall` creates an installation tape than can then be loaded through tools in the System Administration application just as a site operator will do. Developers must include this test and load the segment on a pristine system to be sure that all development environment dependencies have been removed. Failure to correctly perform this step increases the probability that problems will be found when this step is attempted at the SSA.

9. *Perform a system test.* Whether the segment has been installed from tape, created by `MakeInstall`, or created through the `TestInstall` tool, a system-level test should be performed to identify any problems with the COE or other segments for which the developer is responsible.

10. *Test segment APIs.* This step applies only to those segments, typically COE-component segments, which contain APIs that other segments will use. A test suite is required for all segments that submit APIs.

3.2.1.3 Segment Submission

Segment submission to the SSA can be accomplished in two ways. MakeInstall can be run to create a tape that is sent by mail or courier to the SSA. Multiple segments may be submitted on the same tape. This approach is required for classified segments, and for segments that are "very large" and so would require a lengthy transmission time if submitted electronically.

An alternative approach, implicit in Figure 3-1, is to submit the segment electronically. Electronic submission of a segment is an automated process of compressing and encrypting the segment, then using Web technology to transmit it to the SSA. The segment must be in the "pre-MakeInstall" format meaning that alterations made during the installation process have not been performed. These alterations are usually done by a `PostInstall` script (see Chapter 5) which may create data files, perform operations based on hardware type, etc.

1. *Compress and encrypt the segment.* The tool `mkSubmitTar` performs this task on a "pre-MakeInstall" format segment. The directory `Integ`, described in Chapter 5, must contain an annotated description of output from `VerifySeg.` If a segment includes any public APIs, a test suite must be

[28] The segment developer does *not* assign a compliance level to the segment. The SSA assigns a compliance level, but the developer is required to do a self-evaluation and provide the results to the SSA. This approach allows the developer to have a good idea of what the compliance level will be before the segment is submitted, and it assists the SSA in assigning the compliance level. In effect, the SSA validates the compliance tests performed by the developer.

included to test each of them. The test must include an adequate range of test cases and the results expected for each test. Details must be sufficient for use by competent testers who do not necessarily already understand either the application or its individual APIs.

2. *Submit the segment.* The tool `submit` does this electronically across the Internet. Multiple segments can be submitted at the same time.

3.2.1.4 Segment Integration

Segments received, whether by tape or electronically, are placed into the software repository, tested in isolation, and then tested as part of the deliverable system. Validation is performed at each step using exactly the same tool set that the developer used during the development phase. This approach allows many integration responsibilities to be performed by the developer with only a need to validate that they were performed correctly when a segment reaches the traditional system integration phase.

The process steps performed from this point on in Figure 3-1 are the responsibility of the SSA, not the developer.[29] They are described here because developers are still an active part of the process in isolating and correcting problems.

1. *Receive segments.* Segments received electronically are placed in an isolated and safe disk directory. Segments received via tape are placed there manually by a member of the SSA configuration management team. The process point of contact is notified that the segment has been received and is in process.

2. *Validate the segment.* `VerifySeg` is run against the segment submitted and the results are analyzed. Discrepancies between the output of `VerifySeg` produced by the developer and that produced by the integrator can occur for a number of harmless reasons. These are reconciled against the annotated results provided by the developer when the segment was submitted. Segments that fail to pass `VerifySeg` or the reconciliation process are rejected and the process point of contact is notified.

3. *Submit segment to the online repository.* Segments that have been validated by `VerifySeg` are compressed, encrypted, and placed in the software repository. Notification that the segment is now in the repository is sent to the process point of contact.

4. *Test segment in isolation.* The segment is loaded on a test system with the minimal segments required for the operational system. If the test fails, the process point of contact is notified with a detailed description of the problem. The segment remains in the repository but it is not available to anyone except the developer.

5. *Assign segment DII compliance level.* Testing performed by the SSA includes a compliance check using the *I&RTS* and *User Interface Specification* checklists. The checklists produced by the SSA are compared against the checklists submitted by the developer (as described in subsection 3.2.1.2). Discrepancies are evaluated to determine the reason, and the appropriate process point of contact is notified of the compliance level assigned to the segment.

6. *Advance segment to test level.* Segments that work correctly in isolation are advanced to the next testing level and are so noted in the repository. The process point of contact is notified and developers needing the new segment are notified that a beta version is available.

7. *Create Configuration Definitions.* Most segments will not be loaded on every platform. One or more configuration definitions that include the segment are established.

[29] The DISA SSA performs these steps for COE-component segments and mission-application segments within DISA COE-based systems. The SSA identified by the cognizant program manager performs them for other mission-application segments.

8. *Perform system test.* Configuration definitions including the segment are loaded onto platforms for system testing. Those that fail are retained in the repository and a list of problems is sent to the process point of contact. Depending upon the severity of the problems, the segment may be rejected, provisionally made available for other developers to continue working, or accepted with known problems.

9. *Accept segment.* Segments that are deemed to be sufficiently stable are advanced in the test process and declared to be ready for delivery to operational sites. This is so noted in the repository and notification of acceptance is sent to the process point of contact. The segment catalog is updated to reflect that the segment is now available and interested parties (operational sites, program managers, developers) are notified of the new capability.

3.2.1.5 Segment Installation

Segments can be distributed to sites either electronically or by other distribution media as appropriate. The `MakeInstall` tool is used to extract segments from the repository and write them to tapes or other media. The media is then manually delivered to the site. Once received at a site, the site administrator can use the installation tools in the System Administration application to load segments directly onto individual platforms. The installation tools also allow the site administrator to designate one or more platforms as *segment servers*, load segments from electronic media onto the segment server disk(s), and then load platforms across the site LAN from the segment servers. This greatly reduces installation time because multiple platforms can be loaded simultaneously from disk rather than serially from much slower storage media.

Installation can also be performed electronically through the `RemoteInstall` tool. The `RemoteInstall` tool operates in either a "push" or a "pull" mode. In a push mode, the appropriate SSA initiates electronic transfer of segments from the repository to operational sites. Segments can be installed in a push mode to either a segment server or to an individual platform. In a pull mode, the remote site initiates the segment transfer. This is done by selecting the `RemoteInstall` tool from the System Administrator application. Operating in this mode, the `RemoteInstall` tool establishes a connection to the repository, provides the operator with a list of segments that can be downloaded, and provides the operator with the option of loading segments onto a segment server or installing them directly onto a platform.

The discussion of installation given here is necessarily abbreviated. The capabilities provided by the COE are much more powerful. Refer to the appropriate SDMS (Software Distribution Management System) documentation for more information.

3.2.2 Processes Specific to Database Segments

When developing a database segment, the following additional issues pertaining to its database must also be addressed. The discussion that follows specifies requirements of all COE-based database segments. The DII COE SSA will ensure compliance for all database segments for which DISA is responsible. For all other database segments, compliance assurance is the responsibility of the cognizant DOD SSA.

3.2.2.1 Segment Registration

Both the use and the source of data will be identified as part of the Segment Registration for any database segment. The Segment Registration document will also include space requirements for the database within the DBMS (including index space), database scalability (including rate of growth, if any), and application usage. The application usage section must define access for each individual application at the data object level in terms of objects accessed and the mode (read or read/write) of that access. Developers must

identify any COTS tools they are using if those tools will require runtime components to be installed outside the segment.

Where application segments that use databases are being developed separately from the database segment(s) they access, the developers must define the application's required access to database objects. The database segment owner (development sponsor or DISA) controls an application's access to database objects and must approve or reject a segment's proposed read/write access to the database. Database segment owners are responsible for defining generic read access permissions for their databases. In either case an application's access requirements are the basis for defining its corresponding database roles.

DISA will review database segments' contents for duplication of data objects and sources that already exist in common databases or in other database segments. The space or storage requirements of the segment will also be reviewed in the context of storage availability on DII Database Servers. DISA (or the cognizant DOD program manager) may direct developers to use common or external data objects.

Registration of database segments requires additional information to that given in subsection 3.2.1.1:

Functional Area	DOD functional area as defined in DODD 8320.1.
DBMS	The DBMS that is used to manage the segment(s) being registered.
Database/Data Store Name	The identifying database name(s) and/or file name(s) for the segment(s) being submitted, with version numbers assigned in the field.
Using Applications/Systems	The name and brief description of applications or systems known to use the data segment(s) being registered.
Domain Description	A brief description of the information domain of the segment(s) being registered.
Fielded Sites	A general description of the DOD locations where the data segment(s) will be used.

The additional information listed here must be updated when the database segment is actually submitted. It is captured and made available to authorized users for purposes of potential reuse of the data assets provided in the segment.

3.2.2.2 Segment Development

The segmentation process for database segments begins with identifying the database segments that will be established to create the database. It is possible that the database can be implemented as a single segment or multiple segments.

A database segment is the building block that provides specific data services for one or more DII COE applications. If a database is only supporting one application and has no domain tables (e.g. Country Codes), then it may make sense to implement a single application-unique database segment. However, if the database supports more than one application, then the developer should determine whether the database should be implemented with one or more shared database segments. The advantage of multiple shared database segments is that the segments are more granular and a shared data server can be configured to support the data requirements of mission applications without having to carry superfluous data services. A disadvantage of multiple shared database segments is the management of database object dependencies that can be created by such things as foreign key constraints. These inter-segment dependencies complicate the management of segment installation and, moreover, the removal of segments.

Another consideration for developers is to determine which parts of a database are shared between applications and which parts are unique to a given application. From a configuration management point of view, one segmentation strategy would be to place the application-unique database components into a separate segment.

Additionally, determining the contents of a database segment requires several factors to be considered:

- Which tables can be conveniently managed as a unit,
- Which tables are defined to support a functional area,
- What are the sources of data, and
- What are the database object dependencies.

The structure of the database or databases in the segment must be fully described during this phase. This descriptive information includes tables, elements, indexes, privileges, triggers, etc. The database description will be maintained in the `ReleaseNotes` for the database segment. The storage structure of the segment must also be defined.

Developers should examine the SHADE repository of universal and shared database segments for potential reuse. For example, if the SHADE repository contains a universal database segment for country codes, then it may be possible to remove the country-code table from the proposed segment. It is more than likely that the physical schemas will not match but it is possible that the proposed segment could implement a database view to the country-code table in the universal segment.

Any changes to the DISA-defined configuration of the COTS DBMS must be requested from DISA as COE environment extensions. The DISA DII COE Chief Engineer will review such requests to ensure they do not conflict with the needs of other segments, and will be responsible for changing the COTS segments. DISA's defined DBMS configurations are available from the COE Online Services.

A test database is required for all database segments. The purpose of the test database is to allow the SSA to test the segment's operations and to test the applications that access the segment's data. The test database must be unclassified. If the segment's data fill is classified, but the schema is not, a separate data segment must be provided for the classified fill.

3.2.2.3 Segment Submission

Developers must remove all data files from the `DBS_files` directory before submitting the database segment. The files in this subdirectory are the ones owned by the DBMS and used to hold the online database. They are created on a database server during `PostInstall`, and should not be included with the segment. See Chapter 5 for an explanation of the `DBS_files` directory and for more information on the database segment's structure.

Database segments submitted to the DII COE SSA will be included in the SHADE repository.

3.2.2.4 Segment Integration

Developer testing of all applications that access a Database Segment's data structures is performed during this phase to ensure proper functioning and performance of new and existing applications. Following this, a segment installation test will be conducted.

Testing a database segment must include tests of all applications that access the objects in that segment, whether or not they were provided by the database segment's developers. The purpose of this testing is to identify application problems so their respective developers can initiate corrective action.

3.2.2.5 Segment Installation

Database segments are installed only on a database server. Where data fill is a part of a database segment, its installation via `RemoteInstall` may not be supported because of the potential need to transmit a large quantity of data electronically. Network transfer of large data sets can take a long time. Since the

DBMS must be operating in its maintenance mode during a database segment install, users could be denied database services for a significant, possibly intolerable, period of time.

3.3 Migration Considerations

The preceding section dealt with the development process as if it represents new development. However, much of the present and planned functionality is derived from existing legacy systems, not new development, and it simply is not feasible in many cases to totally abandon a system and start over. A migration strategy must be implemented which allows legacy systems to take advantage of COE benefits. The strategy must simultaneously balance full DII compliance versus implementation cost, rapid system deployment versus risk to system stability, porting functionality versus new development, and preservation of capabilities users already have versus duplication.

With the exception of subsection 3.2.1.2, the process outlined in the preceding section applies directly to both new development and migration strategies, or requires minimal customization. However, subsection 3.2.1.2, which describes the segment development phase, requires a few additional special considerations.

It is helpful to remember that the overarching approach is to build on top of the DII COE, not to decompose the COE into constituent parts to build on top of some other architecture or body of software. In other words, the approach is to integrate components *from* legacy systems *into* the COE, not to integrate the COE into an existing legacy system. This perspective is fundamental to successful integration.

The key to reusing the COE and to achieving DII compliance is the concept of the public API. APIs represent the gateway through which segments may gain access to COE services, including the kernel. Software developers and integrators must use public APIs and avoid dependence on a particular version of the COE since the public APIs will be preserved as the COE evolves. Applications *must* migrate away from private or legacy APIs since they will not necessarily be supported in subsequent COE releases.

Given this perspective of integrating components from a legacy system into the COE, the following considerations will lead to a successful migration strategy. The recommendations are not listed in any particular order or priority because what will be an effective sequence will vary from one legacy system to another.

- *Create a requirements matrix.* The matrix should identify requirements already met by the COE, requirements that the COE meets but which require modification, and unique requirements. This matrix represents the development work that must be performed. Modifying COE functionality requires negotiation with the DISA DII COE Chief Engineer and approval by the DISA COE CCB. Mission-unique requirements may be met by porting legacy components, by other mission segments external to the COE, or by COTS products.

- *Identify anticipated source code changes.* Most segments should be able to achieve Level 5 compliance without any source code changes. This is because most of the Level 5 requirements are simply good, standard programming practices (e.g., not hardcoding absolute pathnames in the application). However, above Level 5, source code changes are likely to be required to migrate the legacy system to use COE services. Commercial products are available which will analyze source code and identify API usage and hence help pinpoint areas where changes may be required.

- *Identify public COE APIs to be used.* The API analysis from the preceding recommendation can be useful in determining what APIs from the COE are going to be needed. An initial step in migrating to use COE services might be to create an interim layer that maps legacy APIs to their corresponding COE APIs. This will often help in rapidly achieving Level 6 (Intermediate DII Compliance) from Level 5 (Minimal DII Compliance).

- *Identify areas where the proposed application overlaps the COE*. Runtime compliance at or above Level 6 is largely a process of removing duplication.

- *Identify support services within the legacy system.* These support services are candidates for replacement by COE services and should be partitioned away from the mission application through modularization of the code.

- *Develop a schedule and strategy for achieving Level 8 compliance (Full DII Compliance Level).* Intermediate steps to achieve a lower level of compliance are very useful as progress milestones in the migration strategy.

Segments must demonstrate Level 7 compliance (Interoperable Compliance) prior to acceptance as an official DISA fieldable product and must show migration to Full DII Compliance unless the segment will be phased out.

- *Determine how the segment will be integrated with the Executive Manager.* The COE installation tools provide "hooks" to allow segment functions to be accessed from either an application icon available from the desktop or as options in a pull-down menu within an application. The *User Interface Specification* contains guidelines for which approach is most appropriate for segment features. The Executive Manager uses a commercial CDE product, so consulting CDE documentation will be very useful.

- *Determine which account group(s) the segment will belong to.* Chapter 2 explains that account groups permit dividing users into groups based on how they will use the system (system administration, database administration, etc.). This is important because it is the account group that determines the runtime environment for a segment. The COE allows a segment to belong to multiple account groups because some segments, such as a Printer segment, are of general utility while others, such as a propagation-loss tactical decision aid, are much more specific to a mission-application domain.

- *Determine the required runtime environment extensions.* The COE enforces the principle that segments may extend a base environment according to a set of well-defined rules, but may not alter the environment in a way that adversely impacts other segments. Chapter 5 elaborates on the rules for how segments may extend the environment. The important points here are that segments *must* separate the runtime environment from software development preferences, and identifying changes required in the runtime environment is the key to achieving Level 3 (Platform Compliance) compliance.

- *Negotiate new APIs or modifications with the DISA Engineering Office.* Identifying functionality missing from the COE or required modifications can often serve to drive COE development. Modification of APIs and the introduction of new APIs requires approval by the DISA DII COE Chief Engineer and by the DISA COE CCB.

- *Use only public APIs.* Use of private APIs or APIs from a legacy system may be expedient for an interim period. However, use of such APIs will limit compliance to Level 6 or 7, or lower, and the risks associated with the fact that such APIs are not supported and may vanish in subsequent releases of the COE are the responsibility of sponsors of such a legacy system.

4. DII COE/SHADE Database Concepts

SHADE uses database segmentation and Shared Data Servers (SDS) as the primary underlying mechanisms to enable data sharing. Packaging data into database segments and installing them on an SDS allows multiple organizations and functions to share single copies of a DBMS (i.e., one per physical data server) in the same way that packaging applications into segments allows software systems for multiple functions to coexist on the same platform. At the same time, identifying database segments as Shared or Universal allows explicit sharing of data among software systems, applications, and their user communities. Database segments provide a convenient way for organizations to load SDSs with the data structures and values that users require. This packaging technique will also make it easier for data administrators to collect required metadata on implemented structures and to physically reorganize (multi-segment) databases in ways conducive to distribution and replication.

The function of a COE SHADE SDS together with the databases it manages is to provide information to users through applications that access the databases, and to support system and database administrators' maintenance functions. The operations of an SDS involve the database server, the databases/database segments managed by it, and the applications that access one or more databases. The discussion that follows addresses the operational roles of each.

An SDS provides data management services to its client applications. In order to be useable, it must constitute a stable, reliable operating environment that developers can design for. Database services include tools to support the management, by a site administrator, of users' discretionary access to databases based on the applications they are permitted to use. This is governed by the following principles.

- Users will not need access to all applications.
- Applications will have multiple levels of database access that can be granted to users.
- When access to an application is granted to or revoked from a user, the corresponding database permissions are also granted or revoked.

SHADE database services within the COE are implemented as a federation of application-owned (Unique) and common (Shared and Universal) database segments. Application segment developers control the data and structures that are specific to their Unique segments and can change the data or their structure when necessary. The configuration of Shared and Universal database segments is controlled by the appropriate joint configuration management authority. All these databases reside on the SDS that provides services to the applications, acting as database clients, within the network. The databases within a particular SDS are isolated from each other, physically and logically, by being placed in separate storage areas and by being owned by different DBMS accounts. Database developers sustain this isolation by defining one or more database accounts to own their data objects and by allocating those objects to the owner accounts they have created.

This configuration, using a disk controller and drive analogy, is shown in Figure 4-2. The core database configuration, containing the DBMS Data Dictionary and associated system information, is part of the COE and is represented by the System Database. All other databases, whether provided by DISA, a developer, or some other agency, are included as 'component' databases under the management of the SDS. The set of component databases available from a particular SDS is determined by the set of applications that server's database is expected to support.

From the DBMS perspective, all databases are shared assets, whether they are common or not, because they are accessed by multiple concurrent users. They are also dynamic because their data changes even if their structure remains static. Databases may be interdependent. Databases depend on the COTS DBMS service and are built within its constraints. Databases can be accessed by applications other than those written by the database developer. While database applications today are usually written by the database's developers, this will be less true in the future as SHADE data object reuse increases.

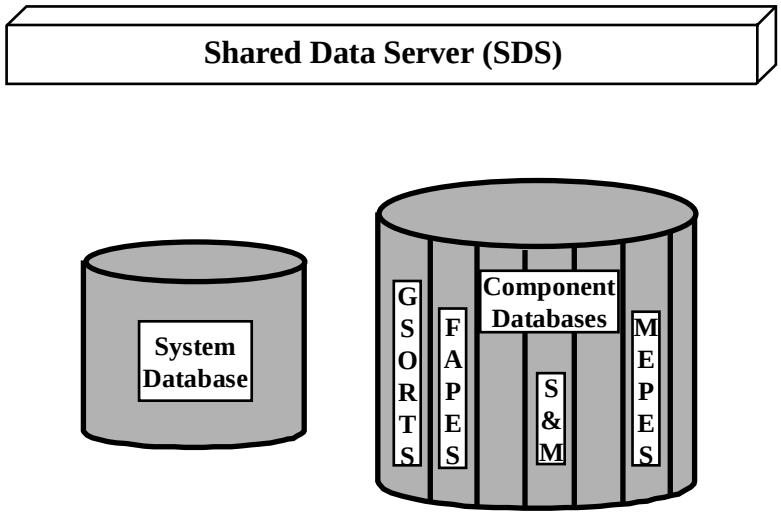


Figure 4-2: Shared Data Server Architecture

Applications that use databases to manage their information are the interface between users and the data. Some applications use their databases interactively, in a transaction-processing mode, to perform the work for which they have been designed. Others have a single process that writes data for many readers. Some pull data from remote sources directly to replace existing data. They then allow read-only access to that remotely provided information.

Users connect to the SDS through the client applications, possibly in multiple sessions. Each session must behave as if it is isolated from the rest of the system and knows of no data other than that belonging to the application it is executing.

Note: The DII COE requires that database transactions implement strict two-phase locking. Two-phase *locking* and two-phase *commit* are *not* synonymous. Two-phase locking is implemented by the DBMS to sustain the atomic properties of transactions. Two-phase commit is implemented in a distributed DBMS to ensure consistent updates of replicated data records. It is used when a distributed database requires synchronous updates. For example, GCCS uses an asynchronous distributed transaction model and therefore cannot use two-phase commit.

4.1 Constraints on Database Developers

The developers of databases and applications accessing databases must conform to the COE database server environment so they do not bypass its features. Conformance also limits the likelihood of data corruption. The combination of the SDS configuration and the developers' implementations must ensure two things. First, each connection of a user to a database through an application must function in the proper context for that application and database. Second, each user's connection to a database must not interfere with any other user's connection to the same or any other database.

The development and integration standards for COE databases support an evolving configuration of database services. Using GCCS as an example, in version 0 each GCCS application had its own database and database management system. Commencing with GCCS 1.1, the separate database servers were replaced by a single-server running a single-instance of the database management system. Each application retained ownership of its database within that instance, but shared the DBMS service with the other applications' databases. The next step was to have a database segment on the server that is accessible from multiple application databases. For example, suppose two application databases need a country-code table. In the prior step, each database would have its own version of the country-code table, which might be identical. In this step, a single copy of the country-code database segment would be on the SDS and accessed by both applications. Thus, the SDS provides shared, concurrent access to multiple databases and database segments with varying degrees of autonomy. COE-based systems are to follow the same approach as that pioneered by GCCS.

The principal reason for this change in GCCS was to meet DOD's information availability requirements. The multiple instance configuration split information among data applications that were uniquely configured to support the needs of specific mission applications. The single-server, single-instance data management service provided by GCCS conserves system resources by not requiring multiple copies of the DBMS to be executing and eases system management by providing a single point-of-entry for database management services. That single point of entry also simplifies application development. However, that is not the only method for implementing SHADE data services. COE systems may implement a configuration that distributes the database over a LAN/WAN for survivability or to distribute the processing load. Multiple DBMS instances may be used for data isolation or to separate different user communities on the same server. Regardless of the specific database server configuration, SHADE requires that information be treated as a DOD corporate resource, not something owned by applications. The benefits that come with the central service does limit the freedom of developers by requiring that they implement databases compatible with the larger multi-database environment. In addition, the increased complexity of a multi-database system could overburden the operational sites' system and database administrators unless it is implemented consistently. This again limits developers by constraining their databases to function within a consistent administrative framework.

The principal consideration for developers is that their applications and databases no longer have exclusive use of the database management system. Instead of being an application-specific data management tool, the DBMS is a central service that supports all applications' databases. As a result, developers cannot customize or tune the DBMS to the particular behavior of any single application. Any such modifications to the DBMS will inevitably affect other applications and databases. Similarly, the individual component databases are no longer the sole occupants of the DBMS. Developers must implement their applications, constraints, and component databases so that they do not interfere with others sharing the same DBMS. Further, because there are multiple databases in the DBMS, applications can connect improperly to other databases. Developers must ensure that their applications connect only to the database they intended to use. They must also design their databases to maintain their own integrity without reference to external applications.

In order for component databases to plug into and play properly on an SDS, they must conform to the standards defined herein. The objective is to support the independent development of maintainable databases that will function reliably within the larger multi-database system. This release of the *I&RTS* has extended the COE tool set to include tools that deal specifically with integration problems related to multi-database environments.

Developers must implement their databases such that the operational sites' administrators can manage the collection of databases. If system and database administrators are required to manage multiple databases, each with its own integrity rules and access methods, their jobs quickly become impossible.

4.2 Database Integration Requirements

The SHADE Database Server is the COE component that provides shared data management within COE-based systems. Regardless of the COTS DBMS used to provide database services, its functions within the system remain the same:

- Support independent, evolutionary implementation of databases and applications accessing databases
- Manage concurrent access to multiple, independent, and autonomous databases
- Maintain integrity of data stored in the DBMS Server
- Provide discretionary access to multiple databases
- Sustain client/server connections independent of the client application's and database server's hosts
- Support distribution of databases across multiple hosts with replicated data and with distributed updates
- Provide maintainability of users' access rights and permissions
- Support backup and recovery of data in the databases.

In addition, database services within the COE are not restricted to a single vendor's DBMS. As a result, developers must implement their databases such that dependence on any particular DBMS vendor's product is limited. The discussion that follows provides more detail on each of these general requirements.

4.2.1 Evolutionary Implementation

The goal of evolutionary implementation is to be able to incrementally develop, field, and improve software and information services. This "build a little, test a little" philosophy applies to databases as well as applications. In the database context, the objective is to field the latest and best information structures and contents, and to progressively reduce the number of structural variants representing the same entities and relationships. Databases and applications should be able to evolve independently in principle, but in practice this is tempered by the dependence of applications on the database's structure. In addition, component databases are dependent on some unique DBMS features for their implementation.

Database developers can still support evolutionary implementation by maintaining the modularity of their component databases. To achieve this goal, component databases must first coexist within the server without corrupting each other's data. This does not simply require isolating databases from each other; it requires that all actions across database boundaries be intentional and documented. The COE/SHADE architecture requires that segments not modify other segments. The same applies to component databases modifying or extending other database segments. When a database segment does have a dependency on some other component database, that dependency will be kept in a separate segment.

Component databases are dependent on the DBMS used for the SDS. The specific commands used for their implementation within the DBMS and the environment it provides are both defined by the DBMS vendor. Database developers must be careful in their use of vendor-specific features so they do not create unintended dependencies on specific database management systems or, more importantly, particular versions of the DBMS, while still taking advantage of the database server's capabilities. To accomplish this, developers shall separate DBMS-specific code from that which is transportable. See Appendix F of this document for information on vendor products. Additional information and guidance on SHADE-specific issues can be found in the SHADE Architecture and related documents.

The same constraints on databases also apply to applications accessing those databases. Application developers must ensure that applications connect through regular, documented APIs and shall not assume the use of particular DBMS versions. This does not prohibit developers from designing to the current version of a COE-compliant DBMS, using vendor-supplied tools that are part of the COE, or from accessing objects in other database segments. It prohibits developers from embedding DBMS vendor's runtime libraries or environment variables in the application segment. For example, developers should not provide their own `coraenv` script in the application segment because it creates an implicit version dependency on that version of the Oracle RDBMS. In addition, this example interferes with the Database Administrator's (DBA) management functions.

The key to managing the evolution of component databases and the applications that use them is documenting their interrelationships. Applications' dependencies on databases shall be documented so that database-segment version changes can be tested with the applications. The component database's dependency on the DBMS will also be documented for the same reason. If developers use DBMS vendor-supplied tools to implement applications, the dependency on the tools will be documented. When applications or component databases access data objects belonging to other component databases, the dependency among the databases shall be documented as well. These dependencies are documented under the `Database` and `Requires` descriptors of the segment's `SegInfo` file. See Chapter 5 for more information.

When one developer is responsible for both applications and databases, the management of such interdependencies is simplified. Database segments and associated application segments will usually be delivered at the same time and installed together. When separate developers are responsible for databases and applications, however, careful coordination between the two developers will be required. As the database federation evolves, it is likely that component database segments will be upgraded before applications that access them. When applications are affected by component database segment modifications, legacy views may be provided as directed by the cognizant authority for the segment. Such views will be read-only, but can allow query tools to continue to function until they are modified to work with the re-engineered database.

4.2.2 Database Segmentation

Another issue with respect to modularity is that of subdividing a database into coherent segments. If a database is only supporting one application, then it might make sense to implement a single Unique database segment. However, if the database supports more than one application then the developer should determine if the database should be developed with one or more Shared database segments and Unique segments. The advantage of multiple shared database segments is that the segments are more granular and an SDS can be configured to support the data requirements of mission applications without having to carry superfluous data services. A disadvantage of multiple shared database segments is the management of database object dependencies such as foreign key constraints. These inter-segment dependencies complicate the installation and deinstallation of database segments.

In the course of determining which data objects will be grouped into a database segment, developers need to consider several factors:

- Data Objects that can be conveniently managed as a unit,
- Data Objects that are needed together to support a functional area,
- Common sources or providers of data,
- Data object interdependencies, and
- Frequency of update.

Modularity can be enhanced by allocating data objects among Shared and Unique database segments. A Shared database segment contains data objects that are intended for use by multiple applications or other data stores. A Unique segment's objects are specific to the applications contained in a specific software segment. Additionally, the developer should investigate the SHADE repository for existing Universal and Shared database segments for potential reuse. For example, if the SHADE repository contains a Universal database segment for country codes, then it may be possible to remove the table(s) defining country codes from the proposed segment and use the existing country code database segment. In the case of a legacy system, a view may need to be created to the Shared or Universal database segment until the application can be modified. Divide the remaining tables in the database

along functional boundaries to form segment groupings (i.e., neither unique nor replaceable by an existing database segment). Potentially Shared database segments should be registered in the SHADE repository. The outcome of this process should be a set of one or more database segments with their corresponding groups of identified database objects. Specific guidelines for creating database objects are found in subsection 4.3.

The database objects in a Shared database segment are common to many applications residing in different segments. Shared database segments prevent duplication of widely used or required database objects, such as reference tables, and procedures, such as validation or conversion routines. They also support interoperability at the data level by standardizing key cross-reference fields. The objects in a Shared database segment must be accessible to many applications, regardless of which database they reside in, and may support other database segments that are then dependent on that Shared database segment. Such segments will often provide generic read-only or read/write database roles (see subsection 4.3.5) to support their use by other segments. In this context, the only distinction between a Shared and a Universal segment is the organizational level at which their contents are managed.

The database objects in a Unique database segment are not open to, nor intended to support, multiple software or data store segments, but are used only by a particular software segment. This software segment owns, controls, and depends on its own database segment, and no other software segment does. Thus, a Unique database segment usually contains the database tables, triggers, and procedures that support specific, intrinsic functions of a software segment; it has no data of value to any other segments.

Dividing data in this manner simplifies the system integration effort. When a change is made to a Shared database segment, all developers of applications that access that segment must be notified and must be given time to adjust their application segments. Otherwise the Shared database segment must incorporate legacy views to support the applications until they can be modified. Changes to Unique database segments, however, require no coordination as only the applications in the dependent segment are affected. In addition, legacy views are seldom required as the applications and their database segment, both usually maintained by the same developer, will be modified at the same time.

Developers should also consider the frequency of updates against data tables when defining their database segments. Separating static reference tables from those that are dynamic allows more flexible system and database administration. The separation may be accomplished by placing the static objects in their own database segment, or by creating static objects in a separate storage area (e.g. an Oracle tablespace for read-only tables) within the segment. The appropriate method will depend on the target DBMS. See subsection 4.3.2 for storage allocation methods and Appendix F for implementation information specific to each vendor's product.

4.2.3 Managing Multiple Databases

The COE database architecture is a federation of databases with varying degrees of autonomy. Federated means that the component databases share DBMS resources. They process data cooperatively but are not part of an overall schema. They may use Shared and Universal database segments. In some cases they may also share or exchange data. Autonomous means that each database remains an independent entity. Individual databases may be modified or upgraded without reference to others (e.g., segments may be added or deleted to support the functionality of the applications that use the database). Individual database segments within a database may not be changed without reference to others unless they are unique segments. Developers are also responsible for maintaining their own data access and update rules.

The federated architecture provides the same modularity within the SDS that mission-application segmentation does for the user interface. The set of databases available from any particular SDS is tailored to the information needs of the individuals using that server. Database segments that are not needed can be omitted. This may appear to conflict with SHADE's stated goals of improving interoperability through data standardization. In fact, it supports those goals by separating the data that can be shared from those that should not be. As a result, developers and data stewards can concentrate their efforts on standardizing those data where there is the most benefit in terms of quality and interoperability.

In order for this to work, each component database must be implemented in a self-contained manner. This is not to say that a database supporting a set of applications should be self-sufficient. One goal of SHADE's modular database implementation is to limit the redundancy of information among component databases. Developers should not incorporate information in component databases that is already available from other, existing database segments. Instead, being self-contained means that each component database must contain all information needed to manage its objects and maintain their integrity. The issues involved in implementing this are discussed in the next subsection.

4.2.4 Data Integrity

Data integrity addresses the protection of the information stored within a database management system. There are three general circumstances that must be addressed.

1. The prevention of accidental entry of invalid data.
2. The security of the database from malicious use.
3. The protection of the database from hardware and software failures that may corrupt data.

Implementation of appropriate data integrity measures is the responsibility of the database developers using the features of the DBMS.

The SDS is responsible for preserving the integrity of each component database and for preventing connections between an application and data that belong to any other application. COE-based systems may well be secure systems that contain and process classified data. The database management component must conform to the security policies and practices of the overall program. Otherwise, the SDS supports the data access restrictions and integrity assumptions incorporated in each database.

The SDS provides the basic functionality expected of a DBMS. It ensures the recoverability of failed transactions or of a crashed system. The atomicity, concurrency, isolation, and durability of database transactions are the responsibility of the applications accessing the server. However, supporting these transaction properties is the server's responsibility. Developers must pay special attention to transaction isolation because of the multi-database configuration of most COE-based systems.

Database developers are responsible for defining and implementing the integrity constraints of their databases. The SDS is responsible for enforcing the developers' integrity constraints when they are defined within the database. Application developers must ensure that their applications connect properly to their databases and do not connect improperly to anyone else's database segments. Adoption of these practices protects all applications' data and allows the SDS to maintain all databases reliably.

Within a component database the implementation of data integrity takes the form of what are often called constraints and business rules. In the current context, constraints are defined as the rules within the database that govern what values may exist in an object. Business rules are those rules within the database that govern how data is updated and what actions are permitted to users.

Until recently, commercial database management systems were limited in their ability to support the variety of constraints and business rules that may be needed in a database. As a result, most constraints and business rules of legacy DII databases have been implemented in the applications, not the database. Because of the federated database architecture and because the applications that maintain those databases are also developed independently, it is difficult to ensure uniform and consistent enforcement of those rules and constraints by a DII COE SDS.

To avoid problems with constraint enforcement in the DII environment, developers must place their business rules and constraints in their databases rather than their applications. This keeps control of data maintenance access in the hands of the developers where it belongs and ensures that constraints cannot be bypassed. Developers have the knowledge of their constraints and business rules; DBAs and users do not.

The reason for placing constraints in the database is shown in Figure 4-3. Application One and its associated component database were implemented with business rules and constraints in the application. Application Two placed those constraints in the database. When a third application (Browser) accesses both databases, it is unaware of Database One's business rules because they are inaccessible. If this application, which could be a user-developed query tool, modifies Database One, it could corrupt the database out of ignorance.

Placing the business rules and constraints in the database promotes client/server independence. The efficient implementation of constraints and business rules will have to make use of the DBMS capabilities. If these rules are in the component database, the application is less dependent on the COTS DBMS product. Also, this approach can reduce network communications loading by allowing the DBMS, rather than the application, to enforce the rules within the database. Checking rules within the database avoids passing multiple queries and their results over the network between the DBMS and the application.

4.2.5 Discretionary Access

Discretionary access addresses the selective connection of users to databases through applications. Database access is discretionary because not all users have the same permissions to use applications. The objective is to ensure that users' database connections operate in the proper context for the applications. Users must be able to operate several different applications at the same time. The DBMS server must effect each application's accesses to different sets of data objects. This means permission to access to specific tables and the mode (read or write) of that access. Because several databases exist on the SDS, each application must be written to access only the database(s) it belongs with; it must be unable to access tables belonging to some other application for which it does not have access privileges. Each user-application connection will have only the permissions needed for that context.

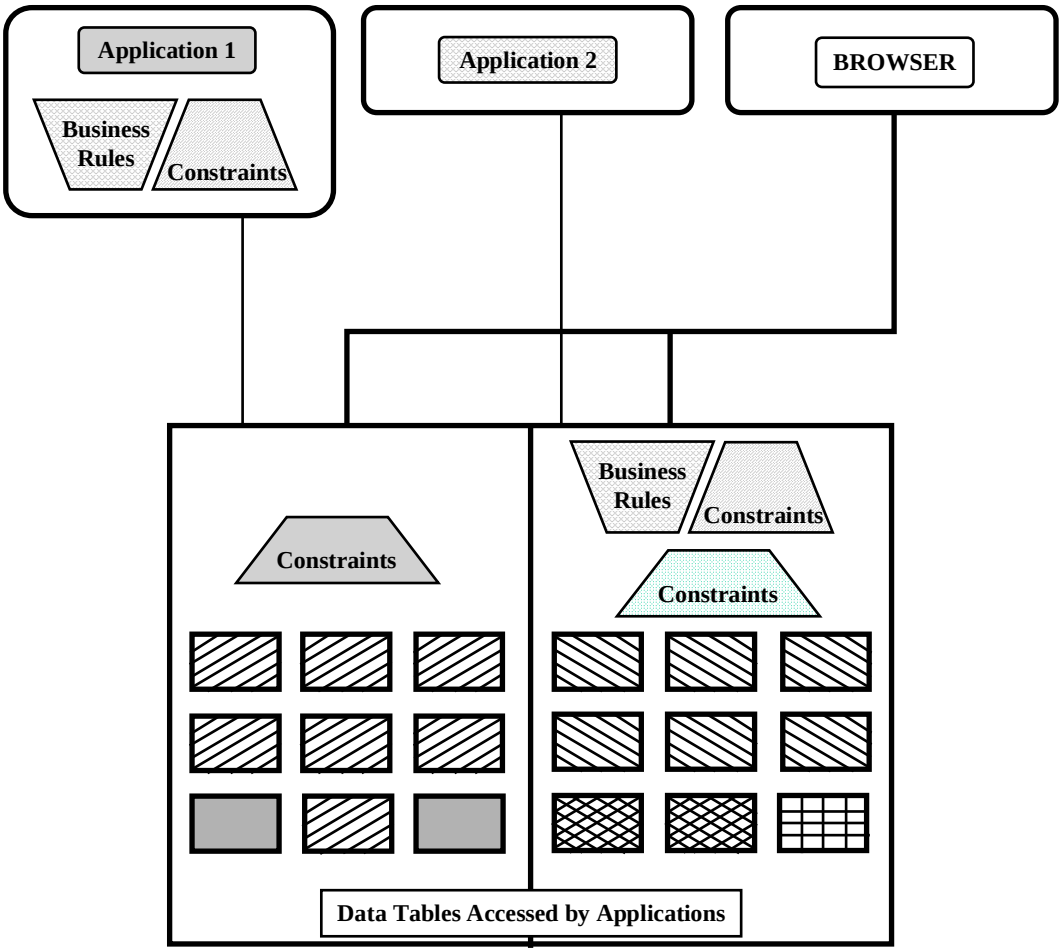


Figure 4-3: Business Rules and Constraints

In this context, DII databases can be broadly characterized as either public or private. A *public* database is intended to be generally available and, in most cases, access to it will be given to all users of a particular system. Public databases are usually read-only. Access to a *private* database is discretionary, not general, and is usually restricted to a small group of a system's users. That system's administrators must specifically grant individuals access to a private database. Private databases often have users with read/write permissions and users with read-only permissions. A public database will be composed of Shared database segments as defined in subsection 4.2.1 A private database may also contain Shared database segments, but the data itself needs to be more closely controlled than the public database. The public/private and Shared/Unique categories address different issues. The distinction has to do with user access in the former case and with configuration control in the latter case. A Shared database has many developers writing applications against it; its schema cannot be easily modified. A public, application-owned database would have one developer but many users; its schema could be changed without affecting other developers.

There are three components to the discretionary access issue: Session Management, Discretionary Access Control, and Access Management. The first refers to the DBMS' ability to keep different connections separate. The second addresses the context of an individual connection. The third deals with the requirements of system and database administrators to manage the accesses that are provided to users. Without the correct functioning of all three components, data integrity and consistency can be compromised.

In order for a COE system to be useable, it must provide support to systems administrators as they manage users' discretionary access to subsets of applications and databases. This means that the approach taken in supporting access management must fit with overall system administration and security policy.

4.2.5.1 Session Management

A *database session* is an individual connection between an application and the database management system. It is the means by which the SDS isolates one user's activities working with an application from all other users that are connected to the DBMS. In this context autonomous applications such as message processors are also database users.

The SDS is responsible for session management as shown in Figure 4-4. In this example, two users are connected to the SDS. The first has two sessions with application A and one with application B. The second has a session with application B and one with application C. The SDS maintains five separate sessions. Two sessions are connected to component database A, two to component database B, and one to component database C; no session is connected to component database D.

Note that each different execution of an application is considered a separate session and is functionally isolated from other executions of the same application. Thus, when User 1 starts two separate instances of application A, the DBMS treats them as different sessions (A1 and A2). This ensures that changes being made in different sessions propagate correctly and do not corrupt data accessed by other sessions.

The key points with respect to session management are that the DBMS, in managing connections, provides sessions to isolate each one from all others. Isolation facilitates transaction management and system recovery. It also supports the traceability of database transactions to the user and application.

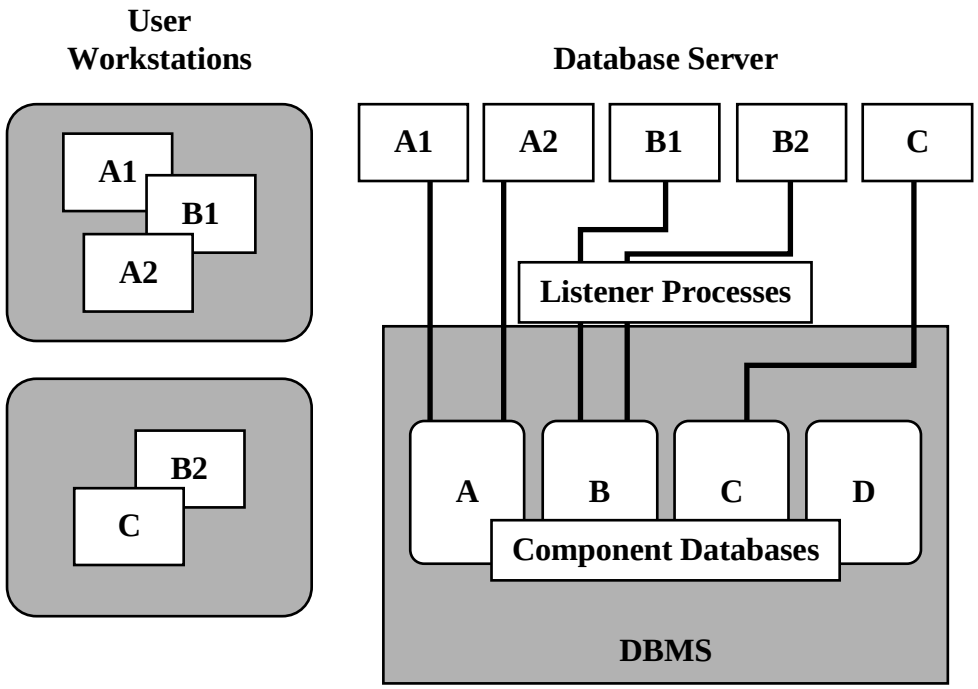


Figure 4-4: Session Management

4.2.5.2 Discretionary Access Control

Discretionary access control is used to manage users' permissions to employ applications to access or modify data managed by the SDS. It has a broader scope than information security. Security is focused on whether users are permitted to know about and allowed to view certain information. Discretionary control of access deals not only with users' permissions to change information but also the context in which they are permitted to make changes.

Users will have access to multiple databases through many different applications. Their overall database permissions are the union of the permission sets of the individual applications they have the right to use. At any point in time only the subset of those permissions relevant to the active session can be allowed to be active.

Figure 4-5 illustrates the need for discretionary access. A user has three database sessions active, one with each of three different applications. Each application accesses a different set of objects within the database. The data objects shown represent all objects that a user has permission to access and are marked to show which application context is relevant to that access. If all of the user's database permissions were active at all times, it would be possible for one or another of the applications to access and modify data that is not relevant to it. Instead, each application must only be able to access its corresponding data objects.

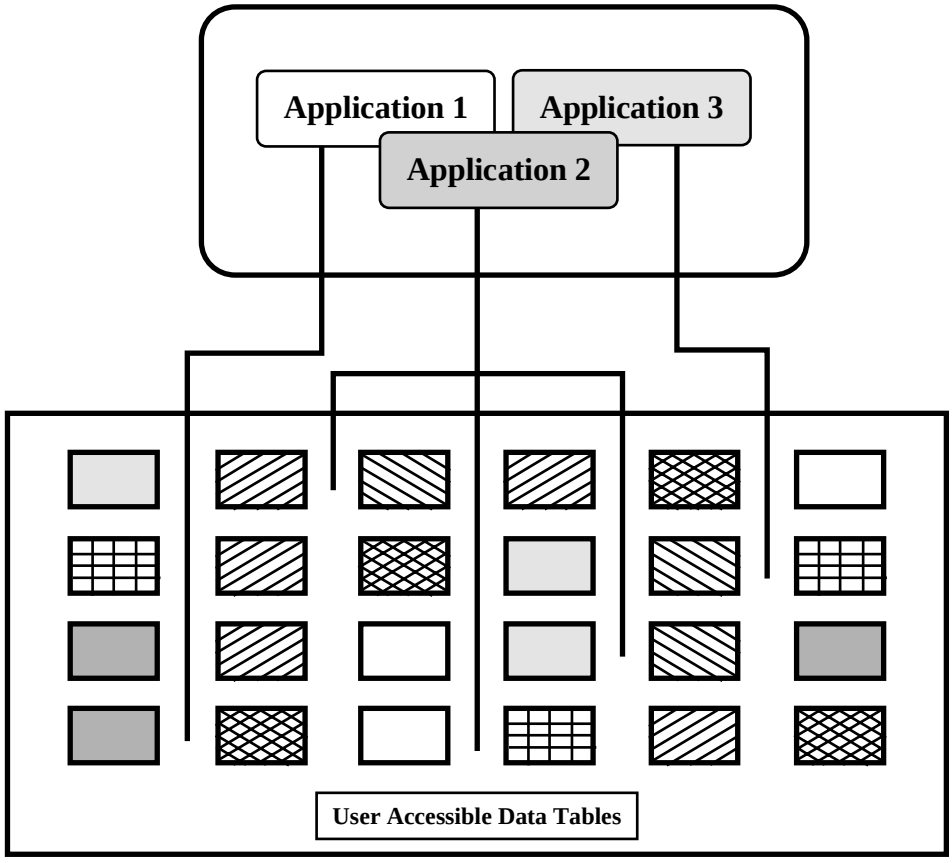


Figure 4-5: Functional Context

It is the responsibility of database and application developers to provide discretionary access controls. The operational sites' administrators are responsible for using those controls when assigning database and application privileges to users. The necessary access controls will be defined in the database segment design as discussed in subsection 4.2.5. Session management by the DBMS provides the database and application developer the isolation needed to implement discretionary access. When designing access controls the following principles apply:

- Users shall have unique accounts within the DBMS. Those accounts shall have only the database permissions needed for their work

- A user's database permissions will only be active within the context of the current application and database session. In other words, when a user starts a database session through some application, that session will only be able to access the data objects appropriate to the application and the only active permissions on those objects will be those appropriate to that application's use of those objects.

These context-specific controls are necessary because users will have access to multiple applications and each application has its own set of database permissions. As a user is granted access to data objects based on the applications needed, the total set of database grants for that user expands. The DBMS manages sessions at the user account level, so each user has all granted permissions on all objects whether they are relevant to the current session or not. If access were not dependent on context, users could have inappropriate permissions for a particular session. For example, a user might be able to write to a database segment that is supposed only to be read by the current application. Such pathological connections to data objects will, almost inevitably, lead to data corruption.

The context in which an application operates on the database is the application's "database role." A *database role* is the minimal set of database permissions needed for an application to function correctly. Since these roles are linked to the application, their definition is the responsibility of the application developer. However, the roles are implemented within the database so they become part of the database segment. Role implementation is discussed in subsection 4.3.5.

4.2.5.3 Access Management

Access management addresses the work of system and database administrators giving users the permissions they need. They must be able to connect users to applications, to databases, and to database segments. They must also be able to revoke or modify those connections as users transfer or assume different responsibilities. The large number of applications and databases available within COE-based systems could make the administrators' tasks unmanageable if access management is not supported with the proper tools. This section discusses the developers' responsibilities for supporting access management.

The act of adding an application to a user's access list, menu, etc. entails adding associated database permissions to that user's DBMS account. Similarly, revoking access to an application requires that corresponding privileges be revoked within the DBMS. Users must have the proper permissions on both the application and the database, so the two system activities are interdependent. Access to applications will often be granted in logical sets or groups of related applications. As discussed in the previous section, access to databases must be linked to each individual application or the functional context of the application is lost. One application could have multiple permission sets if the same executable is used for both read-only and read/write accesses.

The grant association process is illustrated in Figure 4-6. A user is being given permission to use three applications. As a part of that process, the user must also be assigned the database roles associated with those applications. Through the database roles, the user receives the permissions on the data objects needed to use the applications. If, later, the user no longer needs to use these applications, the administrators can reverse the process. When the application permission is revoked, the database roles are withdrawn from the user. The other reason for managing database roles at the application level can also be seen here. Assume that these applications represent a group that is accessed together and that have identical database permissions. If the grouping of applications changes at some point, the collective role might not be valid. In addition, if there is not a direct one-to-one correspondence between applications and database roles, it becomes impossible to determine when a database role should be revoked.

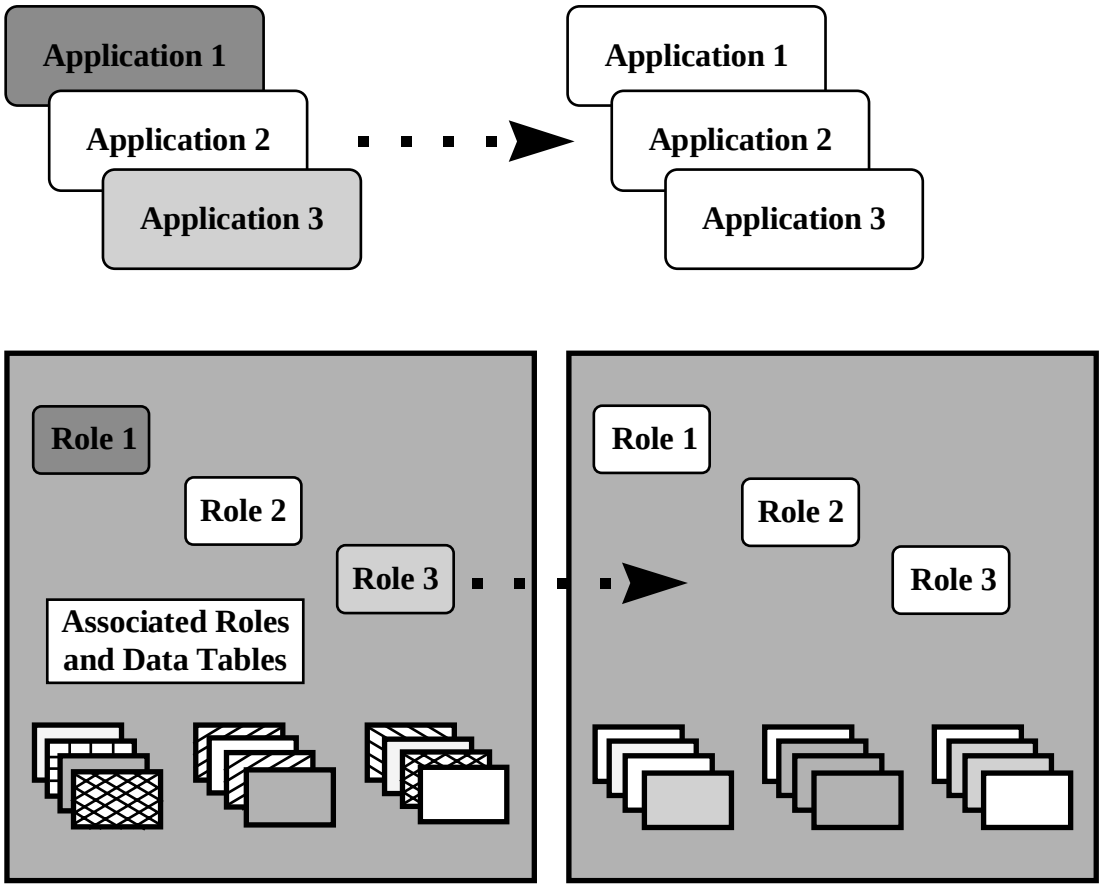


Figure 4-6: Grant Association Process

Database application developers are the only ones with comprehensive knowledge of interactions with the database. They must define the database roles and provide the scripts or command sets that create them for inclusion in the database segment. The scripts that grant and revoke database roles are part of the application segment. This allows them to be executed by the system's administrators when they are managing access to the applications.

4.2.6 Supporting Multi-Database Tools

Access to multiple databases is one of the major benefits that the COE brings to its users. Database browser tools, such as APPLIX, allow users to construct *ad hoc* queries that span different subject areas and that are not supported by mission-specific applications. At the same time, however, such multi-database applications present special problems in the COE context. If the databases were read-only, browsers would not cause problems. However, many databases are designed to be maintained interactively using the applications associated with them. This means that users will have permission to write to databases. Those write permissions are potentially active when a user is using a database browser that means that the browser tool can also write to COE/SHADE databases. This is the reason for the database roles discussed above. Since the browser is independent of the applications designed for particular databases, it will be unaware of any constraints or business rules that are in those applications. Thus it could corrupt data due to its ignorance of the rules.

The key to ensuring database integrity in this case (as in all others) is the enforcement of constraints and business rules within the database, not within the database applications. If the rules are part of the database, they cannot be bypassed. While the SDS may withhold write permissions from browser tools, maintaining the constraints in the

database provides an extra measure of protection. This also supports the future employment of browser tools as multi-database read/write applications.

The second issue is one of understanding the context of a particular database. When users formulate queries that span multiple databases, they are likely to encounter differences in the way information is represented among those databases. This could lead the users to draw erroneous conclusions from their query results because they do not understand the differences between the databases. To limit the chances of this, component database developers shall provide comprehensive information on their databases to be incorporated in the DBMS data dictionary and the SHADE repository. This information is part of the database segment. At a minimum, developers must provide comments for each data element and object (including triggers and stored procedures) that explain its usage and (where appropriate) units-of-measure.

4.2.7 Client/Server Independence

The COE uses a client/server architecture. This applies to database services as well. Developers must preserve the independence of their applications, functioning as DBMS clients, from the SDS. Specifically, applications that access databases must not be built so that they have to reside on the SDS in order to work correctly. It cannot be assumed that all operational sites will have a local SDS. Further, where sites have a local SDS it may be on a separate machine that is dedicated to the DBMS, or the server may be collocated with the application on a single machine acting as the application server and the SDS. To maintain independence and support the client/server architecture, applications cannot assume they reside on the SDS.

To sustain the independence between DBMS clients and the SDS, developers must not mix extensions to the DBMS with their databases and must separate the database from the applications that use it. If specialized data management services are needed by particular applications and are not part of the COE database services, the provision of such services must be approved by and coordinated with the DISA Chief Engineer.

For example, assume some application needs a COTS expert system shell to manage a knowledge base that is a component of the application and that interacts with the SDS. The expert system shell, to work properly, has to be collocated with the DBMS. The expert system then becomes a segment that is separate from the application that uses it.

4.2.8 Distributed Databases

A distributed database is one whose data is spread across multiple sites. Data is replicated in a distributed database when copies of particular objects or records exist in more than one of those sites. Data is fragmented when they are split among sites. Databases are distributed (fragmented or not) to improve responsiveness and increase availability in systems that serve geographically dispersed communities. Databases are replicated to enhance their survivability in the same circumstances. In either case, one implementation objective of any distributed database is to provide location transparency. This means that the user need not know where data is located to be able to access or work on them.

Depending on the component database, the COE/SHADE has several flavors of distributed databases. Some current databases are relatively static and are replicated at multiple sites, but exist independently. They are updated through the periodic replacement of information at each site that has a copy. Others, such as the JOPES Core Database, are dynamic and are replicated concurrently across several sites for survivability. They use transactions to effect updates at the affected sites. Some systems, like the Air Force's Theater Battle Management Control System (TBMCS), both replicate and distribute data on multiple servers within a site to distribute processing and enhance survivability.

The COE provides distributed database management services for the developers of distributed databases so they can maintain location transparency and distributed transaction processing. The specific services implemented for a particular COE database system will depend upon the nature of its distributed data and are the responsibility of that system's Chief Engineer. GCCS, for example, uses an asynchronous transaction model. A financial system may require the use of the more restrictive, but synchronous, two-phase commit.

The technology that supports the distribution of databases as used in the DII COE is evolving rapidly. The GCCS program, for example, does not at present prescribe a particular implementation method. Developers of distributed databases must coordinate their activities with the DII COE Chief Engineer and their program's Chief Engineer to ensure that their approach can be supported and is consistent with the objectives of the broader program. When a distributed database is implemented, developers should keep in mind that the distribution plan (fragmentation schema) may change over time. Distribution methods and the tools used to support them will also evolve as technology matures. Where developers are assigned responsibility for database fragmentation schemas, each fragment shall be in a separate segment so different schemas can be implemented.

The distribution of data also means that users may have access to multiple SDSs. The assignment of users to servers will depend on the distribution schema as implemented for the various sites. The sites' DBAs are responsible for aiming users' processes at the correct SDS. Developers shall not assume that users are attached to a particular server. Developers' applications shall not modify the user's DBMS environment to associate them with a particular SDS. The COE/SHADE data access tools facilitate transparent access to distributed data.

4.2.9 Backup and Recovery

Database backup and recovery address the protection and preservation of information in SHADE databases for DII COE-based systems. The current discussion addresses only those backup and recovery issues specific to databases and database management systems. Conventional system backup and recovery are addressed with the appropriate common support tools for the system as a whole.

COTS DBMS software provides sophisticated tools to prevent data loss or corruption due to system or media failure. Their tools are focused on transaction management with the overriding goal of ensuring that the database can be recovered to a consistent state no matter when or how failure occurs. In the most general sense, DBMS recovery mechanisms maintain transaction logs that keep a continuous record of all database changes. In the event of a system failure, those logs are applied to the database to remove incomplete transactions and recreate committed ones. In the event of media (disk) failure, the last archived copy of any affected DBMS files is used together with the transaction logs (archived and online) to 'roll forward' or recreate all changes that are not in the restored DBMS file.

Execution of DBMS and database backup and recovery is the joint responsibility of a site's system and database administrators. DISA provides tools to assist them in executing their duties. COE developers must implement their databases within the constraints of the DII COE tools and the DBMS vendor tools. Support for special requirements, such as off-line archiving of transaction logs, must be coordinated with the sponsoring COE program office and the DII COE Chief Engineer.

4.3 Guidelines for Creating Database Objects

This section provides guidelines for developers in creating their database segments. Its objective is to support consistency across different databases and improve the mutual independence of the database federation. Also, the guidelines strive to make sure database segments do not inadvertently affect each other.

Developers should strive to make all object names meaningful. Names must start with a letter of the alphabet and may include letters, numbers, and underscores. Names may be 1 to 30 characters in length (except for table names that are restricted to 1 to 26 characters) and cannot be a DBMS reserved word (refer to Appendix F for a list of these reserved words). Case does matter when creating names in the DII COE environment. While a specific RDBMS (such as Oracle) may not be case sensitive in naming objects, there are some (such as Sybase) which are case sensitive. To ensure consistency and portability of database objects and their elements, database object and element names must be implemented in uppercase.

4.3.1 Database Accounts

Three categories of database accounts have been defined within the COE: DBAs, Owners, and Users. They have different functions and levels of access to the DBMS based on those functions.

4.3.1.1 Database Administrators

The Database Administrator (DBA) accounts have access to all parts of the DBMS. They are to be used only for system administration. Their use by database segments is prohibited except during the installation process as discussed in Chapter 5.

4.3.1.2 Database Owners

The Database Owner (DBO) accounts are the creators and owners of the data objects that make up an application's database segment. The name must be unique within the COE community and approved by the SHADE Chief Engineer to avoid naming conflicts. Developers will normally use the segment prefix or a variation of it as the owner account name. The segment prefix will also be used as the database schema name and will be incorporated in database file names as discussed below. Owner accounts must have their password changed after a database installation. Users shall not use the owner accounts to connect to databases. Developers shall not grant the DBA privilege to owner accounts.

All of the database segment's installation, except the definition of physical storage, the creation of the DBO, and the creation of database roles, must execute using the DBO account and password. After a successful installation of the data store segment, the DBO account's password must be changed and its connect capability must be disabled.

4.3.1.3 Users

User accounts belong to the individuals accessing COE databases. Each individual must have a unique user account. User account naming conventions are defined by the individual COE program office (e.g. GCCS Chief Engineer) and will usually be the same as the user's operating system account. The user's account name must be unique within a specific COE database server and may be required to be unique within a COE program. Creation and maintenance of user accounts are a site-DBA responsibility within the rules provided by the specific COE program office. Developers shall not assume the existence of particular users and shall not create user accounts.

The creation of accounts that perform database services is an exception to the rule that developers not create user accounts. Such accounts support autonomous processes, such as message parsers, that access a

database on their own. These processes cannot connect to a database using the DBO account for reasons of security and data integrity, but their identity must be known to developers for their specialized database permissions to be set up correctly. Such accounts will be defined by the developer and created as a part of the segment installation.

4.3.2 Physical Storage

Database management systems provide file management transparency across multiple host computer systems by hiding the details of file storage from the database's data objects. At the same time, however, the placement of data objects on physical storage devices has an effect on system and database performance due to disk contention and other file system access issues.

Developers cannot assume that DII SDSs have uniform hardware configurations. Some will have disk arrays, possibly mirrored, that appear as a single, large mount point; others will have multiple mount points representing separate disks or several mirrored arrays. Further, it cannot be assumed that existing hardware configurations will remain static or that current disk-mirroring technologies will remain in use. DII developers, therefore, shall not use 'raw' partitions, but shall place all files in their segment's directory tree. DISA will provide software tools to insulate developers from the SDS's physical implementation. DISA or the cognizant DOD program office is responsible for providing the core configuration for a COE database server. The site's administrators are responsible for configuring installed servers for optimum performance.

4.3.2.1 Data Store/File Standards

The DBMS-managed components of a database segment can be grouped into functional sets based on their use within the segment. These functional sets are defined as a data store. Data stores are physically kept in database files whose implementation varies depending on the DBMS being used. A segment's database will normally consist of two functional sets (data and indexes) and hence two data stores. The data store identifier will incorporate the database segment prefix and the function of the data store. `GSORTS_DATA` is an example of a data store name.

Developers shall define one or more data stores for their database segments. The objective is to allow data files to be spread across multiple, physical storage devices based on the data store's function within the DBMS. Data store names must be meaningful and use a maximum of 30 characters (uppercase letters, numbers, and underscores). As discussed earlier, the name is case sensitive and only uppercase letters will be used. No DBMS reserved words will be used.

Data store names must also be associated with the segment and function. Most applications will have either two or three data stores: Data, Indexes, and (if needed) Static data. The following naming convention is to be used:

`<segment prefix>_DATA`,
`<segment prefix>_INDEX`, and
`<segment prefix>_STATIC`

for the three storage areas respectively. The Logs within a Sybase database are treated as data stores in a Sybase implementation.

4.3.2.2 Data Storage Implementation

Database segments shall create their data stores through the segment's `PostInstall` descriptor.

Database segments shall use the `COECreateDS` API to implement physical data storage. This API allocates physical storage and creates the data store. For Sybase, this includes the storage area for logs. `COECreateDS` hides the SDS's implementation of physical storage. In this way the database segment is

insulated from the physical server implementation, whether it uses raw devices or file system directories, has disk arrays, or uses other storage techniques. Figure 4-7 illustrates data store allocation.

Where `COECreateDS` is not available, developers will provide the scripts to create their data stores and the operating system files associated with them. Data files will be created in the `DBS_files` subdirectory of the database segment using the API provided by the DBMS vendor. One or more data files may be created to support each storage area. The method for file creation varies with the DBMS being used. See Appendix F for DBMS-specific file creation information. The data file names should be chosen so they are clearly associated with the storage area. The recommended naming convention is

`<segment prefix>_<store type><n>.dbf`

where '*store type*' is the storage area's purpose (e.g. index) and '*n*' is a one-up serial number for the file. An example data file name is `gsorts_data1.dbf`.

CREATE_DS DBSORT < DBSORT_LIST

where DBSORT_LIST file contains:

DBSORT_DATA 1,000K
DBSORT_INDEX 1,000K
DBSORT_LOG 300K LOG

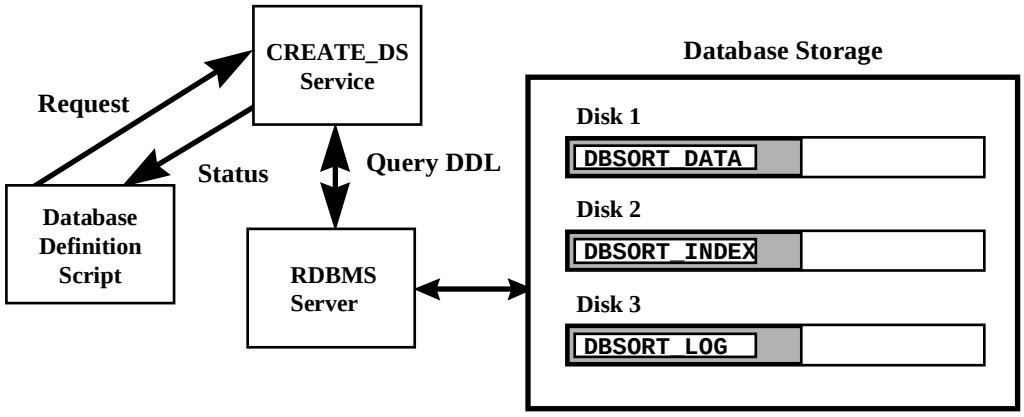


Figure 4-7: Data Allocation

4.3.3 Database Definition Scripts

A *database definition script* is a shell script that contains all database definition commands for a specific database object. These objects include tables, views, triggers, and stored procedures. The name of the script is the same as that of the database object it defines. Depending on the object type, multiple sections can be defined within one file to perform all the data definition functions required for that specific database object.

The scripts used to create data objects are also used by database administrators in the maintenance of the databases and the SDS. DII database administrators have to manage thousands of data objects (tables, views, etc.) spread across multiple database owner accounts. Routine maintenance such as rebuilding corrupted indexes or views can become impossible because the DBA cannot locate the script file that contains the object's definition among the thousands of scripts on the SDS. To avoid these problems, DII

developers must organize their Data Definition Language (DDL) commands into a series of database definition scripts. These scripts must conform to a particular file naming convention and structure.

The database definition script is structured to execute various database definition commands based on the input argument given to it. This functionality is implemented with a case statement that executes on the input argument. Table 4-1 lists the input arguments to be used for database definition scripts. See Appendix F for examples of data definition commands.

A database definition script for a table should contain all constraints, triggers, and indexes for the table. Legacy views (see subsection 4.3.4.3) of the table, if authorized by the cognizant DOD configuration management authority, may be included in the table's definition script. Other views must be created using their own scripts. Rules, stored procedures, packages, and other objects should each be created in their own separate scripts.

Database roles that are associated with a database schema, such as a default read-only role, can be provided with a database segment's definition scripts. The grants for that role, since it is part of the segment, can be incorporated into the table and view definition scripts. Grants to database services accounts (e.g. message parsers) can also be incorporated in those scripts. Database roles that are associated with applications or those whose grants span multiple database owners must be created using their own scripts. These scripts should include all the grants needed for the role regardless of the object's owner. Such grants should be segregated by owner.

The `CREATE_DATA_STORE` argument for a database definition script should only be used when the COE tool `COECreateDS` is not available.

4.3.4 Database Objects

The definition of a database schema – the set of data objects, their interrelationships, constraints, and rules for access or update – is the responsibility of the developers. Developers of application database segments shall not duplicate data objects that are part of the corporate databases provided by DISA. Where possible and appropriate, developers should take advantage of and share objects belonging to other databases as found within the SHADE repository. If a database segment does not meet the full needs of the developer, changes should be proposed to the cognizant DOD configuration authority for the database segment meeting most of the needs. The developer may choose to develop a similar database segment, pending resolution of the change request. To facilitate sharing of data, developers shall provide the definitions of their schema components for inclusion in the DBMS data dictionary as discussed in section 4.1. Definitions for data stores, tables, elements, stored procedures, and views are stored in the system's data dictionary tables as comments. The maximum length allowed for the description is 255 characters. In addition, narrative information on all these databases should be provided during Segment Registration so developers can access their definitions in the COE online services (see Chapter 10).

Developers shall provide DISA with their proposed database schema early in the segment design process. The schema will be reviewed for duplication of objects in other component databases. See Chapter 3 for more information on the database segment development cycle.

4.3.4.1 Database Tables

Database tables are the objects that store data records. Within a database schema, data elements will be logically grouped to form tables. Table names must be meaningful and a maximum of 26 characters in length (uppercase letters, numbers, and underscore). This size differs from the 30 characters available to other objects because of the legacy view naming convention (see section 4.3.4.3). If Oracle database snapshots are being used for data replication services for other sites, developers should limit the table name to 20 characters. Oracle will use the remaining six characters to identify the internal tables and views that support a database snapshot. No reserved words may be used in the table name.

Argument	Purpose
CREATE_DATA_STORE	create a data store (use CREATE_USER to create the DBO account first)
DROP_DATA_STORE	remove a data store
CREATE_ROLE	create a database role
DROP_ROLE	drop a database role
CREATE_RULE	create a Sybase rule
DROP_RULE	drop a Sybase rule
CREATE_TABLE	create a database table
DROP_TABLE	drop a database table
CREATE_VIEW	create a database view
DROP_VIEW	drop a database view
CREATE_CONSTRAINT	create a database constraint (i.e. foreign key)
DROP_CONSTRAINT	drop a database constraint
CREATE_INDEX	create an index
DROP_INDEX	drop an index
UPDATE_INDEX	perform update statistics for Sybase indexes
CREATE_USER	create a database user account
DROP_USER	drop a database user account
DISABLE_LOGIN	Revoke connection privileges (login) from a database account
ASSIGN_GRANTS	assign grants to a user or role/group
REVOKE_GRANTS	revoke grants from a user or role/group
LOAD_DATA	load a table with data (from within the command script)
CREATE_PROCEDURE	create a stored procedure or database package
DROP_PROCEDURE	drop a stored procedure or database package
CREATE_TRIGGER	create a database trigger
DROP_TRIGGER	drop a database trigger
CREATE_SEQUENCE	create an Oracle sequence
DROP_SEQUENCE	drop an Oracle sequence
REGISTER_DATA	load application data (i.e., configuration parameters)

Table 4-1: Definition Script Arguments

The creation of tables in System-owned storage areas (e.g. the Oracle SYSTEM tablespace or the Sybase master database) is prohibited. The tables must be created in storage areas created by and belonging to the application database segment. When creating a table, the storage area name must be specified. “Create table” statements must stipulate `NOT NULL` or `NULL` constraints for each column because different DBMSs may default differently on this constraint type.

4.3.4.2 Data Elements

Data elements are the columns or fields within a schema that are grouped together into tables. Data element names shall comply with DOD standards from the DOD Data Model (DDM) and Defense Data Dictionary System (DDDS) where applicable. Within a schema developers should use the same characteristics (data type, length, number precision, default values, constraints, and definition) for all occurrences of the same element name. If elements are chosen from the DDM, they shall use the data type and units of measure prescribed in the standard.

Developers shall not use data types that are machine-dependent. This applies primarily to numeric data. Data elements may be shared across tables and data stores, and across COTS DBMS servers. As an example, the ‘float’, ‘double’, and ‘real’ data types are machine-dependent in both Oracle and Sybase. See

Appendix F for more information on data types available in specific DII COTS DBMS and which are machine independent.

The use of default values and declarative constraints is recommended to ensure data integrity and consistency. Developers must balance this against instances where invalid data items must be forced into the system, especially when dealing with real-time data. In such cases declarative constraints could cause data loss.

4.3.4.3 Database Views

A view does not actually contain or store data, but derives its data from the objects on which it is based. These objects can in turn be tables or other views. View names must be meaningful and a maximum of 30 characters in length (uppercase letters, numbers, and underscore).

Views are often used to restrict users' access to vertical (columnar) or horizontal (row-wise) subsets of data tables. Views can also be used to hide data complexity when displaying related information from multiple tables or to present data from a different perspective than that of the base table. Views can provide location transparency for local and remote tables in a distributed database, a convenient way of storing complex queries, and isolation of applications from changes in definitions of base tables.

Views can be queried, updated, inserted into, and deleted from, with restrictions. All operations performed on a view affect the base tables of the view. Current DBMSs are limited in their ability to support updates through views. If developers need updateable views, the DBMS's capabilities and restrictions must be kept in mind. If the updateable views are required for security or data privacy, developers should not grant users access to the base tables, only to the views.

In general, the following restrictions apply to updateable views.

- **Horizontal (row-wise) views:** SDSs can support inserts, updates, and deletes through horizontal views. Such views include those where one table is used to constrain the view to a subset of rows in another table. Developers are responsible for implementing appropriate error handling if users try to insert a row that duplicates a hidden row or that contains a value in the restricting column(s) the users are not permitted to see.

- **Vertical (columnar) views:** SDSs can support updates and deletes through vertical views as long as the database constraints do not reference hidden columns. Inserts can only be supported if all hidden columns are allowed to be null or if triggers are provided to populate them with default values. Developers are responsible for implementing appropriate error handling if a user's update violates a constraint on a hidden column.

- **Multi-table views:** At present, the SDSs implemented in the COE cannot consistently support data modifications through views of more than one table. Developers should implement such updateable views in applications. These views should be accompanied by comparable read-only views of the individual tables.

A view is dependent on the objects referenced in its defining query. All of these objects must exist, and the required privileges to these objects must have been granted to the owner of the view before the view is created. Views will be created in a database segment as part of its install process.

Legacy Views are views created to support applications written against earlier versions of a database object. Such views make the object appear as it did in an earlier version of the database segment. They support read-only legacy applications. Applications that need to update data will not be able to use legacy views. Code modifications required to update data in the new data structures will need to be coordinated with data structure changes.

The decision whether to require Legacy Views rests with the DII COE Chief Engineer working with the affected program's Chief Engineer on a case-by-case basis, although developers may choose to do so on their own. In most cases, DISA will require Legacy Views only for Shared or Universal public databases whose tables support a large number of read-only users.

When Legacy Views are provided, they must be implemented in the following manner. When a database table is created, a view that maps directly to the table must also be created. When the database table is modified, the view of its previous version is also modified so that applications accessing the view are unaware of them, and a view that maps directly to the new structure is created. This method allows applications that access the table to continue to operate if immediate source code modifications are not possible. Applications must eventually be modified, but in the meantime views can be maintained to support previous versions of the table. Figure 4-8 demonstrates the use of legacy views across four versions of a database table.

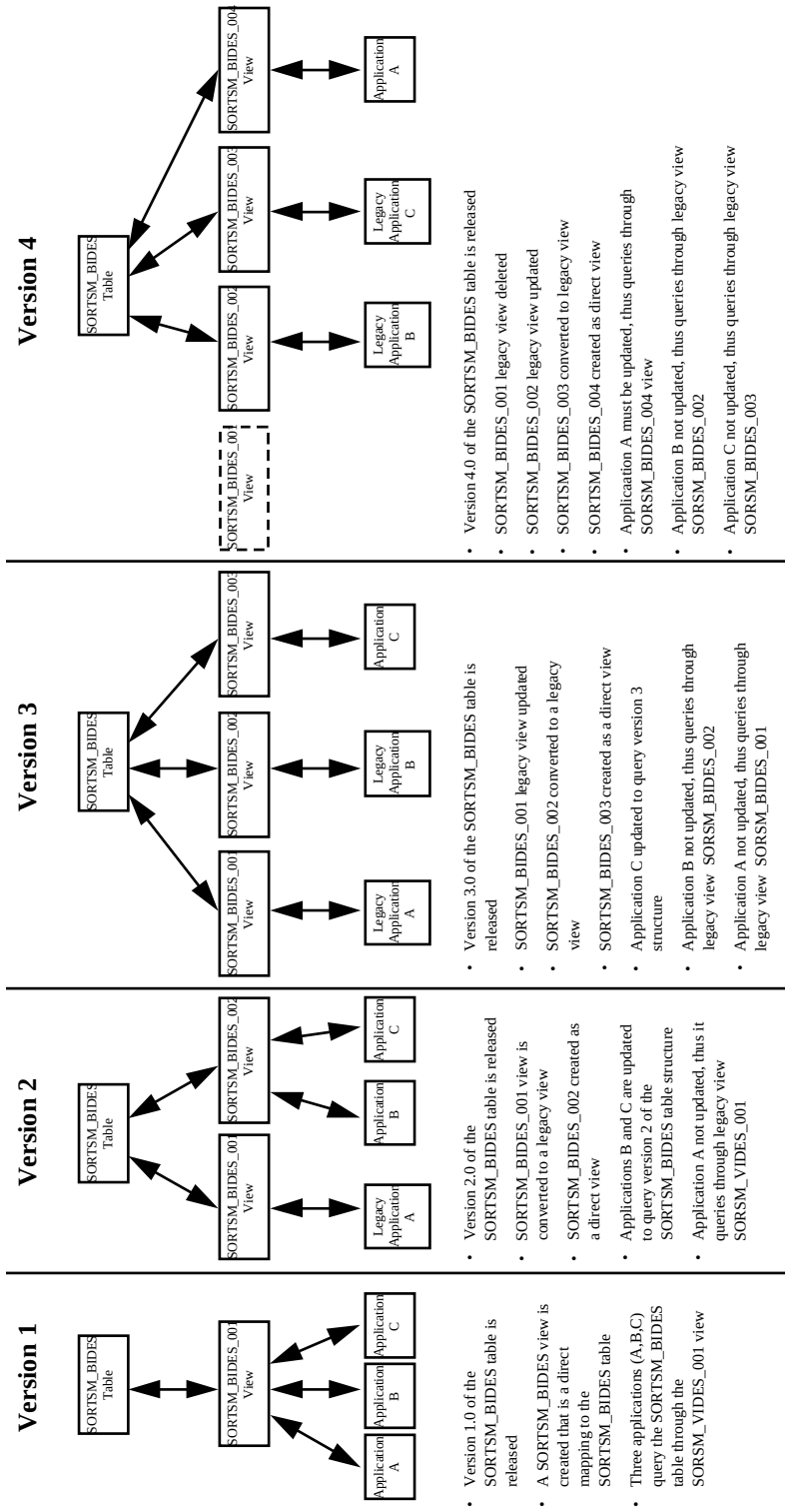


Figure 4-8: Legacy Views

View names for legacy views shall consist of the table name followed by a three-character table sequence number. They will be a maximum of 30 characters (uppercase letters, numbers, and underscore). Example

legacy views are `NID_ACFT_003` and `IDBIND_001`, where the first is a view representing the third release of the NID aircraft table and the second is a view representing the first release of the Integrated Database (IDB) individuals table.

4.3.4.4 Rules on Database Objects

Rules on database objects incorporate several different concepts. Their underlying purpose is to maintain database integrity through the enforcement of the constraints and business rules of the database.

For purposes of this document, the following definitions apply. *Constraints* are restrictions on data elements with respect to the values they may contain. For example, a country-code data element could be constrained to the set of Defense Intelligence Agency (DIA) prescribed two-character country codes. *Business Rules* are restrictions that occur in the context of database operations that affect multiple interrelated objects and elements or that are beyond the ability of a constraint to express them. For example, any update to a facilities table may require that an entry be written to an audit table recording the ID of the user making the change and the time at which it was made.

Within the SDS, developers may use DBMS constraints, stored procedures, rules, packages, or triggers to implement either constraints or business rules. The choice among these will depend on the capabilities of the COTS DBMS being used.

4.3.4.4.1 Constraints

Developers should define all entity integrity constraints and referential integrity constraints that apply to their database schemas. The information in these constraints is vital for maintaining database integrity. Entity integrity should be enforced whenever possible using default, unique values, and check constraints. Domain Keys (e.g. the SQL Check constraint) should be used to maintain the validity of column values. Unique columns should be constrained rather than indexed. While the DBMS may use an implicit index, as Oracle does, to enforce uniqueness, defining the constraint clearly documents the database design for the users. Database primary keys, foreign key constraints, delete cascade actions, and update/delete restrictions should be used to maintain referential integrity.

Primary and foreign keys convert logical relationships that are implicit in the database design into explicit relationships. Primary keys identify unique physical records. Foreign keys relate primary keys to data in other tables by requiring each value in a column or set of columns to match those in a primary key in the referenced table. Foreign key constraints enforce referential integrity by preventing invalid data entry into the database tables.

Where appropriate, constraints should be used to supply default values for columns. The NULL/NOT NULL constraint must be explicitly stated for each column in all tables because different DBMS implementations may behave differently with respect to nulls.

Constraints must be explicitly named. Constraint names must be meaningful and must not use reserved words or default names. They may not exceed 30 characters (uppercase letters, numbers, and underscore). The recommended naming convention is

`<table name>_<cons>`

where '*table name*' is the name or abbreviated name of the table or table and columns involved in the constraint and '*cons*' is PK for a Primary Key, FK for a Foreign Key, or CK for a Check constraint. Foreign key constraint names should incorporate references to both tables. Examples of constraint names are `IDBF_PK`, `EWIRD1_EMIT_FK`, and `ACFT_USR_CTRY_CK`.

In most cases developers will wish to create their constraints after the data fill has been completed in order to speed up the fill process. The implicit index that accompanies a Primary Key or Unique constraint will

slow the data fill significantly. Constraints should not normally reference data objects that are outside the database segment. See below for methods to implement inter-database constraints when they are needed.

Constraints should still be included in a database segment even when they cannot be enforced. If developers must allow invalid data items into their database, as may be the case when processing real-time data, they may not be able to enforce declarative constraints without losing information. Constraints should still be defined, but disabled (e.g., by using Oracle's disable constraint command) so that users and administrators can understand the database schema. The `ReleaseNotes` segment descriptor and the comments on the object stored in the data dictionary shall state that these constraints are deliberately disabled so the site's DBAs know that it is intentional. If constraints must be left disabled, the developers are responsible for providing tools that support cleanup of invalid items.

4.3.4.4.2 Stored Procedures

Database stored procedures and functions consist of a set of DBMS commands (e.g. SQL statements, and Oracle PL/SQL or Sybase Transact-SQL constructs) that are stored in the database and can be invoked by an application to perform a task or a set of related tasks. Stored procedures and functions can be used to obtain tighter control of database access. In addition, they improve performance by reducing the amount of information that travels over a network and because they do not require interpretation prior to their execution. The use of stored procedures and functions also reduces memory requirements as only a single copy is loaded into memory for execution by multiple users.

Stored procedures are used to maintain database integrity or to enforce business rules when the constraints imposed are too complex for simple SQL constraints. These procedures are stored in the database and can be executed from any environment in which an SQL statement can be issued. A maximum of 30 characters (uppercase letters, numbers, and underscore) may be used for the stored procedure name. Procedure names should incorporate the name of the object(s) they modify and some meaningful indication of their functions without using reserved words. Two examples of stored procedure names are `NID_UPDATE_PROC` and `GSORTS_FETCH_UNITID`.

Stored procedures should not normally reference data objects that are outside the database segment. See below for methods to implement inter-database stored procedures when they are needed.

Database stored procedures are installed after all of the database objects defined in the database segment have been installed. In general, the stored procedures in a database segment should support integrity checks that are typically invoked by triggers. A database segment may also provide stored procedures that perform standard access functions against the segment's tables. These access functions can provide better performance and reduce maintenance efforts if underlying structures are changed.

4.3.4.4.3 Triggers

A database trigger is a procedure that is automatically executed when a triggering event occurs on the associated table. A trigger can only be defined on a table and will fire whenever the associated event occurs on the table or a view of that table. The action of a database trigger may cause another database trigger to fire. Triggers can be used to generate derived column values, implement complex security rules, perform auditing, maintain table replication, prevent invalid transactions, and enforce referential integrity.

Most triggers will be used to maintain database integrity. Others may be used to signal or send data to other, interrelated or dependent database segments. Triggers may also be used to support the proper replication of data and to perform data conversions. They are not to be used to start application processing based on data entry. Trigger names must be meaningful (the table name and trigger type should be part of the trigger name) without using reserved words. They may use a maximum of 30 characters (uppercase letters, numbers, and underscore). An example of a trigger name is `TAC_REMARKS_UPD_TRIGGER`.

Triggers should not normally reference data objects that are outside the database segment. Database segments should not install triggers on data objects outside the segment. See below for methods to implement inter-database triggers when they are needed.

Database triggers are installed after all of the database procedures are installed. This order is prescribed because triggers may invoke stored procedures. A trigger's body may contain DBMS commands (e.g. Transact-SQL or PL/SQL blocks) or it could invoke stored procedures to perform the same functions. The use of stored procedures to support triggers is recommended for performance and maintainability.

4.3.4.5 Indexes

An Index is an optional structure associated with a table that is used to quickly locate rows of that table or to ensure that a table does not contain duplicate values in specific columns when a uniqueness constraint cannot be used. Indexes speed up retrieval when applications query a table for a range of rows or for a specific row by providing a faster access path to data. Indexes are logically and physically independent of data. The creation or deletion of an index may occur at any time and does not affect the data stored in the associated table. Furthermore, creation or deletion of indexes only affects the speed of data retrieval, but does not prevent any applications from functioning. Once created, indexes are maintained by the RDBMS and are automatically updated when the data change due to addition, deletion or modification of rows. The presence of many indexes on a table decreases performance when inserting, updating, or deleting data as the associated indexes must also be updated. Indexes also require storage in the DBMS; the use of multiple indexes requires more storage.

Index names must be meaningful without using reserved words. A maximum of 30 characters (uppercase letters, numbers, and underscore) may be used for the index name. It is recommended that the index name incorporate a reference to the table and column for clarity. Developers should review the capabilities of the DBMS before indexing small tables (less than 4000 rows). Indexing small tables can actually hurt performance if the DBMS searches the index instead of reading the entire table into memory. The DBMS's query optimizer may ignore indexes on small tables. Indexes should not be used in place of Primary Keys or Uniqueness constraints.

When considering a column or group of columns for an index, keep the following guidelines in mind:

- Indexes should not be used in place of primary keys or uniqueness constraints. If the DBMS treats nulls in a manner that prohibits the enforcement of these constraints, developers should use a unique index to maintain data integrity.

- To minimize lock/device contention when insertions occur frequently, clustering should always be performed on a key that is statistically more "random" than other keys and is usable in many queries. This is generally not the primary key. Prime candidates for clustering keys include columns accessed by range or used in Order By, Group By or Joins. For example, Date/Time could be a good index key for event data. Long strings generally make poor indexes.

- Too many indexes can hurt performance of inserts, deletes, and updates.

- Prime candidates for non-clustered indexes are columns used in queries when the data being accessed is less than 20% of the data in the column.

- Keep the size of the key as small as possible to improve index storage and data retrieval.

- Indexes help select statements and hurt inserts/deletes. Consider when most of your operations will use the index and, if so, whether the overhead required for the index is worth it.

- Storage of indexes in a separate data store can improve database performance.

- The ordering of columns in SQL 'where' clauses may affect the behavior of the DBMS query optimizer. Check the DBMS vendor's documentation to identify such effects and use the vendor's evaluation tools to assist in optimizing DBMS commands.

- Consider using the various index types offered by the DBMS. B-Tree indexes are good for range selections and ordered retrieval, but can suffer performance problems when used on large sets of ordered, sequentially appended (e.g. time series) data. Hashed indexes are fast, but do not easily support ordered retrievals. Bit-mapped indexes are efficient for binary fields like sex.

4.3.5 Database Roles

A database role, in the general sense, is a group of access privileges for database objects. These roles implement the discretionary access controls discussed in subsection 4.2.5. Database roles also simplify the management of user privileges within the DBMS. They are created by the database segment developers or the developers of applications accessing databases to define sets of access privileges that can be given to users by their sites' DBA. Role names must be meaningful (the database or application name should be part of the group or role name) without using reserved words. They may be a maximum of 30 characters (uppercase letters, numbers, and underscore). Developers should strive to associate roles and their privileges with the applications accessing the database. Each role should have only the privileges needed by the application it supports.

As discussed in subsection 4.2.5.2, active database permissions should be limited to the minimum set needed for the session in progress. Such permission sets are specific to an application's connection to the database. This means that each application requiring access to any database object must have a well-defined database role that includes only the privileges needed by the application and that the role/group be granted only to users who are authorized to run the application. In this case, it is the responsibility of the database segment which supports an application's software segment to create the specific database role for the application and to connect to the DBA account (see the `COEPromptPsswd` API in Appendix C) to assign the grants on the required objects to the newly created role. The DBA account has all necessary privileges to assign grants on any object to any role.

A Shared database segment must provide generic "read-only" roles because of the dependencies of other segments upon it; it may provide "read/write" roles. Developers may create more generic roles or groups that consist of a set of privileges (such as read-only or read/write) on a group of objects (such as all of the objects in a database or some subset of them). Such roles are usually created to provide read-only access to an entire database for users of browsers or query builder tools. In this case, it is the responsibility of the database segment that creates those objects to also create the generic role/group and assign the grants on the required objects to the newly created role/group. Such generic roles are useful when widely used, large, read-only databases such as the NID must be implemented. Such generic database roles should be used with caution as they may grant applications more privileges than they really need. Generic database roles should seldom, if ever, be used to grant write permissions. Database developers who implement generic roles or groups must balance the advantages this type of role against the risks of unnecessary or excessive privileges.

Consider the following example. A database segment named TEST has five tables: MASTER, DATA1, DATA2, REF1, and REF2. Two applications, APP1 and APP2, are associated with the segment. The segment should have two read/write roles, TEST_APP1_RW and TEST_APP2_RW, one to support each application. It could also, optionally, have a read-only role, TEST_RO, for users of browser tools. If only one read/write role were created, then users of APP1 could inadvertently modify data that should only be changed using APP2 and vice versa.

When applications are not developed by the database segment developer, the application developers are responsible for creating the roles required to access the database through their applications. The access requirements for such roles must be defined by the application developers and included in the information

provided during Segment Registration as discussed in Chapter 3. The permissions required by application's database roles are subject to review by DISA and by the associated database segment's sponsor. These roles will be granted the privileges required to run the application. These privileges may include: delete, insert, select, and update for tables and views; and execute for procedures, functions, and packages. Grants of privileges to roles are discussed in the next section.

In order for the privileges on objects to be assigned to a role, the grantor must have permission to do so, and those database objects must exist. When application developers define database roles to support their applications and those roles are not part of the principal database segment, the roles and the grants that enable them become part of a database segment that is dependent on the database segment or segments that create the referenced objects. See subsection 4.3.7 for more information on inter-database dependencies.

Database roles shall not be granted to DBAs. Their administrative privileges already allow them to grant roles to users without owning the roles. The database roles that are part of the COTS DBMS shall not be altered by developers.

4.3.6 Grants

Grants are the permissions on database objects that allow users to access data they do not own. When a database object is first created, the only account that can access its contents is the owner of that object. Users must be explicitly granted permission to access an object. Privileges that can be granted include: delete, insert, select, and update for tables and views; and execute for procedures, functions, and packages. Privileges that should not be granted include index and alter for tables. Grants allow the DBA to administer and the DBMS to enforce the discretionary access controls required. As discussed in the section on database roles, developers should grant only the minimum set of permissions needed for the applications that access their databases. Grants should be made to roles/groups and not to individual users.

Consider the previous example. APP1 is used to create and modify records in DATA1. It uses MASTER and REF1 as lookup tables. APP2 has the same function for DATA2 using MASTER and REF2 as lookup tables. The read/write role associated with APP1, TEST_APP1_RW, should be granted the select privilege on MASTER and REF1, and select, insert, update and delete on DATA1. It should have no privileges on REF2 or DATA2. TEST_APP2_RW, the read/write role for APP2, would have select on MASTER and REF2; select, insert, update and delete on DATA2; and no privileges on DATA1 or REF2. TEST_RO, for users of browser tools, would have the select privilege, only, on all five tables.

Granting data access to DBMS 'PUBLIC' users is prohibited. Granting data-access privileges to user accounts with the 'GRANT OPTION' or granting administration privileges on database roles is prohibited. Developers shall not make grants of application-level permissions to DBA accounts or to database roles used for DBMS administration. Where segments' applications or databases need special permissions on DBMS objects (e.g. query Oracle's 'v$' tables), the developer must request them from the DII COE Chief Engineer. Such grants should be kept in a separate database definition script (to be executed by the DBA) within the database segment that needs them.

4.3.7 Inter-Database Dependencies

Inter-database dependencies occur whenever database objects in a segment are dependent upon objects in some other database segment. A database object is a dependent object if it references any other object(s) as part of its definition. When a dependent object is created, all of its references to other objects must be resolved. If it has dependencies on non-existent objects, the dependent object may not already have been created or it may have to be validated when the objects it references come into existence. The creation of a dependent object may also fail if its owner does not have the appropriate access to all referenced objects. If the definition of any of the referenced objects is altered, the dependent object may not function properly or may become invalid.

Dependent objects that reference objects created, managed, and maintained by the same database segment do not introduce inter-segment dependencies. In contrast, dependent objects that reference objects in other segments do add complexity to the installation, de-installation, administration, and maintenance of a database segment. Before using dependent objects, developers must balance the advantages of dependent objects against the disadvantages of introducing segment dependencies.

Database segments with intersegment dependencies sometimes benefit from smaller storage and reduced data redundancy. Using data objects that belong to other segments frees up the storage that would otherwise be used for replicas of those objects. When replicated objects are eliminated, changes to those objects need not be propagated across multiple database segments. At the same time, having only one copy of a widely referenced table is likely to increase data quality and currency. Eliminating copies of data objects also reduces the processing load on the SDS by eliminating duplicate updates when changes are made.

Such dependencies also affect the modularity and scaleability of the SDS. Dependent segments must be installed after the database segment they reference. Further, as is the case with other segment types, dependencies can easily propagate when placed on segments that are, in turn, dependent on other segments. Furthermore, when inter-segment dependencies are defined, circular dependencies can be created. A circular dependency exists when two segments depend on each other. In such cases, neither segment can be installed because both require the other to be installed first. If a circular dependency cannot be resolved, then the two segments may have to be merged into a single, larger segment or the dependent code can be moved to a third segment. The dependency of one database segment on another segment's data objects could require the installation of multi-gigabyte databases so that one or two of their tables can be used by some other segment.

Because of the tradeoffs involved in the employment of dependent objects, their use in DII systems is subject to review and approval of the DII COE Chief Engineer.

Where inter-database dependencies are needed they shall be implemented such that the object(s) creating the dependency are owned by the database segments that they belong to. This means that a foreign key constraint belongs in the segment with the table it constrains, not in the segment with the table it references. A post-update trigger added to a table belongs in the segment with that table, not in the segment of the table it updates. Such dependencies may have to be placed in separate database segments that modify the segment owning the object that creates the dependency. See Chapter 5 for more information on segmenting databases that have dependencies. The `Requires` descriptor for such database segments must identify all dependencies on other database segments. In addition, the `Database` descriptor must be used to identify the data object(s) being referenced in other segments so that DISA can choose the most effective segmentation strategy for databases that are widely used.

The following sections describe how developers should implement inter-segment dependencies that may occur through the use of dependent objects, constraints, and database roles.

4.3.7.1 Data Objects

Database segments will have dependent data objects (tables or views) when their information needs can be partially satisfied from tables or views contained in other database segments. If an external table fully satisfies the information needs, it should be referenced directly. Developers may use a dependent view to extract subsets of information from external tables or views or to change the presentation of information (e.g. change units of measure or combine columns). Developers may use views to combine internal tables with external objects to provide information supersets. Also, a table could reference an external object either as a source of constraints or, through a trigger, as a provider of data.

Names of objects created in other schemas must identify the inter-database linkage. Otherwise they are subject to the naming restrictions of their object type. Developers are responsible for ensuring that their object's names do not conflict with those already in the schema.

A table will be dependent on another database segment if its constraints reference objects in that other segment or if it is populated or maintained using a trigger based on an external object. Developers may also create a table that is a superset of an external object to avoid creating and maintaining partially redundant objects. That table would then be combined with a view that joins it with the external object. Developers must use an 'outer join' when defining such a view/table combination unless appropriate triggers are created to prevent decoupling when updates occur to either the internal or external table.

A view that references a table (or view) outside its own segment is dependent on the database segment containing the base table (or view). Once such a view has been created, it will become invalid and have to be recreated if its base table (or view) is modified, renamed, or dropped. Any privileges or synonyms on the invalid view also become invalid until it is recreated.

Developers shall not create indexes on objects in other database segments. Indexes have significant impact on system performance. While they speed retrieval of records, indexes slow updates to tables. The effect of uncontrolled index proliferation could dramatically damage the overall functioning of a DII system. If developers desire indexes on tables in other database segments, they must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess the effect of additional indexes. If, based on overall requirements, the request is approved, the segment responsible for the creation of the table will be modified to also create the index(es) required by other segments.

4.3.7.2 Rules in Other Databases

Database segments have dependent constraints or business rules when their integrity constraints or operations involve objects from other segments. Such rules may include foreign keys that reference another schema's tables or triggers that propagate updates based on another schema's transactions.

Any rules – whether they are constraints, triggers, or procedures – shall be created in the schema of the object they are attached to. Names of rules created on other schemas must identify the inter-database linkage as well as the rule's function. Otherwise, they are subject to the naming restrictions of their object type. Developers are responsible for ensuring that their rule names do not conflict with those already in some other schema.

Developers may create constraints in their own schema that reference objects in other database segments. They may not create or modify constraints on objects in other schemas. Such constraints could invalidate otherwise legal updates to the other database. When additional constraints are needed on objects in other database segments, developers must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess the effect of these constraints. If, based on overall requirements, the request is approved, the segment responsible for the creation of the table will be modified to also create the constraint(s) required by other segments.

Developers may create triggers and stored procedures or functions on objects in other schemas as long as they do not modify or update the other database's information and do not change the constraints or business rules of the other database. It is permissible, for example, to use a 'post-insert' trigger to copy data from an external data object to one in the developer's database segment. It is prohibited, by contrast, to use such a trigger to change data in that other segment's table.

Excessive use of triggers can result in complex interdependencies that may be difficult to maintain. When implementing a specific function via triggers, developers must keep in mind that a database transaction will rollback if execution of the associated trigger(s) is not successful. Trigger developers must implement exceptions to handle errors or unexpected results that may occur during the execution of a trigger. These exception handlers must ensure that a trigger fails 'open' and allows the owning segment's database transactions to complete regardless of the processing of the dependent trigger. If additional triggers and stored procedures or functions are needed in other database segments, developers must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess

their effect. If, based on overall requirements, the request is approved, the segment owning the object affected by these triggers, stored procedures, or functions will be modified to incorporate them.

4.3.7.3 Database Roles Spanning Multiple Databases

Developers may need to create roles whose permissions span multiple databases in order to take advantage of their information and to correctly represent applications' information needs. Since database roles implicitly are created at the database server level, which segment they belong to is irrelevant. However, all objects they reference must exist before the role may receive its grants. Accordingly, such roles shall be part of a dependent database segment as discussed in Chapter 5. That segment is dependent on every segment whose objects it references. It must list all of the segments under its `Requires` descriptor. See Chapter 5 for more discussion of the `Requires` descriptor.

This page is intentionally blank.

5. Runtime Environment

This chapter describes the software configuration for the COE runtime environment. All software and data, excepting low-level components of the COE kernel, are packaged as segments. A *segment* is a collection of one or more software or data units most conveniently managed as a unit. Segments are constructed to keep related units together so that functionality may be easily included or excluded.

There are six *segment types* corresponding to the different types of components that may be added to a system:

1. **COTS:** A segment totally comprised of commercial off-the-shelf software.

2. **Account Group:** A segment that serves as a template for establishing a runtime environment for individual operators.

3. **Software:** A collection of executables, shared libraries, and static data that extend the base functionality and environment established by an account group.

4. **Data:** A segment composed of a collection of data files for use by the system or by a collection of segments.

5. **Database[30]:** A segment that is to be installed on a database server under the management of the DBMS and ownership of the DBA. A Database segment can only be installed on a database server and the installation tools enforce this. Note that a database client application segment can be installed on any platform and usually is a software segment type.

6. **Patch:** A segment containing a correction to apply to another segment whether data or software. The corrections entail replacing one or more files.

In addition, segments may have attached characteristics, called *segment attributes*, which serve to further define and classify the segment. There are six segment attributes[31]:

[30] Database server segments are supported only on UNIX servers for this release. Database application segments may be created for either the UNIX or NT environment.

[31] Subsection 5.5.1.10 discusses how to indicate segment attributes with the `SegName` descriptor. Segment attributes are noted by the appropriate parameter within the `$TYPE` keyword of the `SegName` descriptor.

1. **Aggregate:** A collection of segments grouped together and managed as an indivisible unit. (This implies that segments within an aggregate cannot be installed across separate platforms.) The segment whose attribute is indicated as `AGGREGATE` is called the parent and is considered to be the "root" segment. The parent segment name is the name presented to an operator as the name of the aggregate. An aggregate can have only one parent segment.

2. **Child:** A segment that is part of an aggregate, but is subordinate to a single segment designated as the parent. An aggregate can have multiple child segments.

3. **COE Component:** A segment that implements functionality contained within the COE, as opposed to a mission-application segment.

4. **DCE:** A segment that implements either a DCE server or a DCE client application. The DCE attribute *must* be specified for any segment which uses DCE segment descriptors.

5. **Web:** A segment that uses Web-based technology to create the application. A Web segment is either a Web server, or a Web-application segment (e.g., a client application). A user requires a Web browser to access Web-based segments.

6. **Generic:** A segment that is to be automatically added to all "usual" account groups (see subsection 5.4.11). This feature allows a segment to participate in multiple account groups without the need for the segment to explicitly name each account group.

> **Note:** The attributes listed here are often used in the vernacular as if they are segment types (discussion of an aggregate segment, a COE-component segment, a Web segment, etc.). Technically such usage is incorrect because these are *attributes* and not *types*. When discussing segments by attribute, it is implicitly understood that there is an underlying segment type, usually software.

Segment installation is accomplished in a disciplined way through instructions contained in files provided with each segment. These files are called *segment descriptor files* and are contained in a special subdirectory, `SegDescrip`, called the *segment descriptor subdirectory*. Sections within the segment descriptor files are called *segment descriptors*, *segment descriptor sections*, or just *descriptors*. The segment descriptor files embody a technique that allows a segment to "self-describe" itself. That is, the segment descriptor files contain pertinent information describing the segment, such as the segment name and type. This information is used by other software in the COE and other segments that need to access functionality contained within the segment. But the descriptive information is also used by people to aid in the integration process, to aid in security analysis of the segment, or in configuration management. Installation tools process the segment descriptor files to create a carefully controlled approach to adding/deleting segments to/from the system. The format and contents of the segment descriptor files are the central topic of this chapter.

Principles contained in this chapter are fundamental to the successful operation of the COE and achieving DII compliance is largely determined by how well developers apply the details given in this chapter. Appendix B summarizes the compliance requirements stated in this chapter into a series of checklists organized by Category 1 compliance levels. Developers are required to adhere to the procedures described

---

The parent for the aggregate is designated by the `AGGREGATE` parameter. The Child attribute is indicated by the `CHILD` parameter. COE Component is subdivided into the `COE CHILD` and `COE PARENT` parameters. Similarly, the Web attributed is subdivided into the `WEB APP` and `WEB SERVER` parameters. Finally, the Generic attribute is indicated by the `GENERIC` parameter.

herein to ensure that segments can be installed and removed correctly and that segments do not adversely impact one another. Unless otherwise noted, all requirements apply to both UNIX and NT.

Note: In this chapter and throughout the *I&RTS* mention is made of occasions when approval is required by a Chief Engineer. Unless otherwise stated, this means the DII COE Chief Engineer for COE-component segments and mission-application segments that affect interoperability. All other references refer to the Chief Engineer responsible for the mission-application segment (e.g., GCCS Chief Engineer, ECPN Chief Engineer). The Chief Engineer is not necessarily a DISA engineer, and will not be for the majority of the mission-application segments. Likewise, use of the term SSA refers to the responsible SSA unless otherwise qualified.

5.1 New and Obsolete Features

This DII COE release includes a number of improvements over previous COE releases. A list of the more significant improvements is provided here for developers who are already familiar with a previous DII COE release.

The present release is backwards compatible with previous DII COE releases. Segments presently in use do not require modification to work with the features described here. However, certain features from previous JMCIS and GCCS COE releases are now obsolete and support for them will eventually be phased out. Obsolete features are listed in a subsection below.

All of the features from the previous *I&RTS* have been preserved. Segments which have been migrated to any version of the DII COE do not require additional work to be compatible with this issue of the *I&RTS*. Compliance-level requirements have not been increased with this release, but the compliance criteria in Appendix B have been reworded and reorganized for clarity.

Periodic modifications to the DII COE and the *I&RTS* are made for several reasons:

- to address non-UNIX environments,
- to allow extension to other problem domains,
- to provide support for new and emerging technologies,
- to generalize the COE concept,
- to improve site installation and administration of segments,
- to simplify or clarify certain segment descriptor files,
- to further reduce integration problems,
- to meet emerging mission requirements, and
- to apply lessons learned.

5.1.1 New Features

This subsection summarizes new features in this release that were not present in the previous *I&RTS* release. Its purpose is to serve as a handy reference of new features for developers already using the DII COE.

- Database applications are supported through SHADE. Descriptor information is provided in this chapter.

- The concept of data scope (local, global, segment, etc.) is extended to encompass database scope (e.g., unique, shared, universal).

- The draft PC-based COE from the previous *I&RTS* release has been formalized and incorporated as appropriate to this Chapter. It is further described in Chapter 6. Several new descriptors and keywords have been added to support PC NT applications.

- Support is provided to add NT registry entries (see the `Registry` segment descriptor).

- Standard NT file extensions (e.g., `.TXT`, `.EXE`, and `.BAT`) are supported for segment descriptor files.

- Web-based applications are supported and are described further in Chapter 7. Descriptor information is provided in this chapter.

- Guidance and support for DCE applications is provided. DCE-based applications are described further in Chapter 8. A new set of descriptors (`DCEClientDef` and `DCEServerDef`), a DCE segment attribute, and several new keywords are provided to describe DCE segments.[32]

- The `$KEY` keyword is added to enforce certain requests (such as installation with "root" privileges) that require Chief Engineer approval.

- The location for shared libraries is now specified (i.e., in the segment's `bin` subdirectory).

- Child components in an aggregate may now have a conditional load attribute. This is described more fully below, but it allows a child segment to be loaded only if it represents a newer version than what is already on disk.

- The concept of a generic segment is added. A generic segment is automatically made a member of every account group, except those which are character-interface-based. The segment may also specify account groups that it is to be excluded from.

- Support is added for three new types of processes: RunOnce, Privileged, and Periodic. Privileged is available for UNIX only, but the other two are available for both UNIX and NT. RunOnce processes are executed the first time the system is rebooted, but not thereafter. Privileged processes are those which require "root" permissions to execute. Periodic processes are the UNIX equivalent of `cron` processes, permitting a segment process to be run at specified intervals.

- Support is added to allow site installers to temporarily install a segment to test it before installing it on the rest of the system.

- Support is provided to allow site administrators to create application servers that contain software for multiple platform types. Support is included for "dynamic loading" of segments.

- Segments may add executables to run during the user profile creation/deletion just as with the account creation/deletion process. Support is also added to allow executables to be run when a profile switch is performed.

- The segment installer tool, `COEInstaller`, issues a warning to the operator performing the installation if an attempt is made to load a segment that is an earlier version of one that is already on the disk.

- The `COEInstaller` tool maintains a status log of segments as they are loaded and provides the ability to print the status log. The status log may also include output from scripts (such as `PostInstall`) that is normally sent to `stdout` or `stderr`.

- A `$EQUIV` keyword has been added to the `SegName` descriptor. In effect, this allows a segment to be known by an alias.

- The `Help` descriptor has been added as a placeholder for future expansion. Its purpose is to identify "help files" within the segment and their format (UNIX man page, HTML, etc.).

- A "partial segmentation" process is defined (see subsection 5.7) that provides the advantages of the segmentation philosophy but allows a COTS vendor's distribution media and approach to be utilized.

[32] In this *I&RTS* release, DCE servers are available on UNIX platforms only. DCE client applications may be on UNIX or NT platforms.

5.1.2 Obsolete Features

The features listed below are being phased out because changes were required to extend the DII COE to address the Joint community, to address problem domains other than command and control, and to extend to non-UNIX platforms. The previous release of the *I&RTS* indicated most of these items as obsolete. They are collected here as a ready reference. This release adds only one new requirement: usage of the `$KEY` keyword. This keyword is used in instances where the *I&RTS* requires Chief Engineer authorization for some requested feature, such as permission to create a COE-component segment. To preserve backwards compatibility for existing features, `VerifySeg` only issues a warning if the `$KEY` keyword is missing. An error is generated when the `$KEY` keyword is missing for new features. Developers should begin using the `$KEY` keyword in all appropriate places because a future release will issue errors instead of warnings.

Support is still provided for each of the obsolete items listed below, but documentation for them has been removed from this release of the *I&RTS*. Segment developers and program managers should upgrade[33] to the latest DII COE to ensure future compatibility. Support for the obsolete features may be removed from the next release. The tool `VerifySeg` will issue warnings when run against old segments to identify obsolete features.

- The `MACHINE` environment variable is now obsolete. The `MACHINE_OS` and `MACHINE_CPU` environment variables should be used instead. Segment developers should not depend upon `MACHINE` being defined.

- Individual segment descriptor files are now obsolete. The `SegInfo` descriptor file should be used instead. It is divided into sections which correspond to the earlier individual descriptor files. Conversion to `SegInfo` is required for Level 8 compliance.

- Subdirectories `progs` and `libs` are now obsolete. Subdirectories `bin` and `lib` should be used in order to conform to conventional practice.

- The old format of the `Data` segment descriptor is obsolete. The size required is now specified in the `Hardware` descriptor instead of the `Data` descriptor. Level 8 compliance requires uses of the new format.

- Previous versions of the COE allowed `DEINSTALL`, `PostInstall`, and `PreInstall` to run with root privileges. This capability is no longer the default. The `$ROOT` keyword *must* be used instead and Chief Engineer approval is required to run with root privileges.

- Previous releases of the COE allowed a `$PATH` keyword in the `Menus` and `ReqrdScripts` descriptors. This is now obsolete since the *I&RTS* specifies the location of where files must be located relative to the segment's home directory.

- Segment descriptors `ModName` and `ModVerify` have been replaced with `SegName` and `SegCheckSum` respectively. The `SegType` descriptor file has also been replaced by the `SegName` descriptor file.

- In earlier releases, the parent segment for a child had to be listed in the `Requires` descriptor. This is no longer required because by virtue of naming the aggregate parent in `SegName`, there is an implied dependency. Child segments use the `$PARENT` keyword to explicitly name the aggregate parent. The parent uses the `$CHILD` keyword to explicitly name the children in the aggregate.

[33] The obsolete features are primarily in the content and format of the descriptor files and should not require any source code changes. The effort required to upgrade should be a matter of editing the segment descriptor files and running `VerifySeg`. A tool, `ConvertSeg`, described in Appendix C is available to automate the conversion to the extent possible.

- The `$COMPONENT` keyword is now obsolete and is replaced by the `$CHILD` keyword.

- Previous COE releases automatically provided a system menu bar. Applications must now use the Executive Manager APIs to explicitly request a system menu bar.

5.2 Disk Directory Layout

This subsection describes the COE approach for a standardized disk directory structure for all segments. A standardized approach is required to prevent two segments from overwriting the same file, creating two different files with the same name, or similar issues that frequently cause integration problems. Unfortunately, such problems are often not discovered until the system is operational in the field.

In the COE approach, each segment is assigned its own unique, self-contained subdirectory. This subdirectory is called the segment's *assigned directory* or the segment's *home directory*. The segment's assigned directory is established at segment registration time. It obviously must be unique among all segments that are installed in an operational system. A segment is not allowed to directly modify any file or resource it doesn't "own" - that is, outside its assigned directory. Files outside a segment's assigned directory are called *community files*. COE tools coordinate modification of all community files at installation time, while APIs to the segments which own the data are used at runtime.

Figure 5-1 shows the COE directory structure. The root-level directory for the COE is `/h`. Underneath `/h`, disk space is organized into the following categories (note the close parallel to segment types):

COTS	segment descriptors for installed COTS products
AcctGrps	templates for establishing a runtime environment context
COE	component segments constituting the COE
data	subdirectory for shared (local and global) data files
Web	subdirectory for Web-application segments
Segments	one or more subdirectories for mission-application or other segments
USERS	operator home directories with operator-specific items such as preferences
TOOLS	collection of useful tools for the development environment

Web-application segments are collected into their own subdirectory to segregate them from all other types of applications. This is to make it easier to identify and control them from a site-administration[34] perspective. The Web-server segment is a COE-component segment and therefore is located under the `COE` subdirectory. Web-application segments may or may not also be COE-component segments, but they are placed under the `Web` subdirectory in either case. If they are also COE-component segments, the specialized processing performed for all other COE-component segments is done as well. The installation tools automatically place Web segments in their proper location.

Figure 5-1 does not show other important disk directories, such as the UNIX `/etc` directory. The `/etc` directory is one of a family of related directories which contain UNIX system files. Other COTS products may require specific directories as well, and there are other important system directories that are specified to each operating system.

[34] Web servers and mission-application segments will likely be placed behind a firewall to administratively restrict platforms that outside users can gain access to.

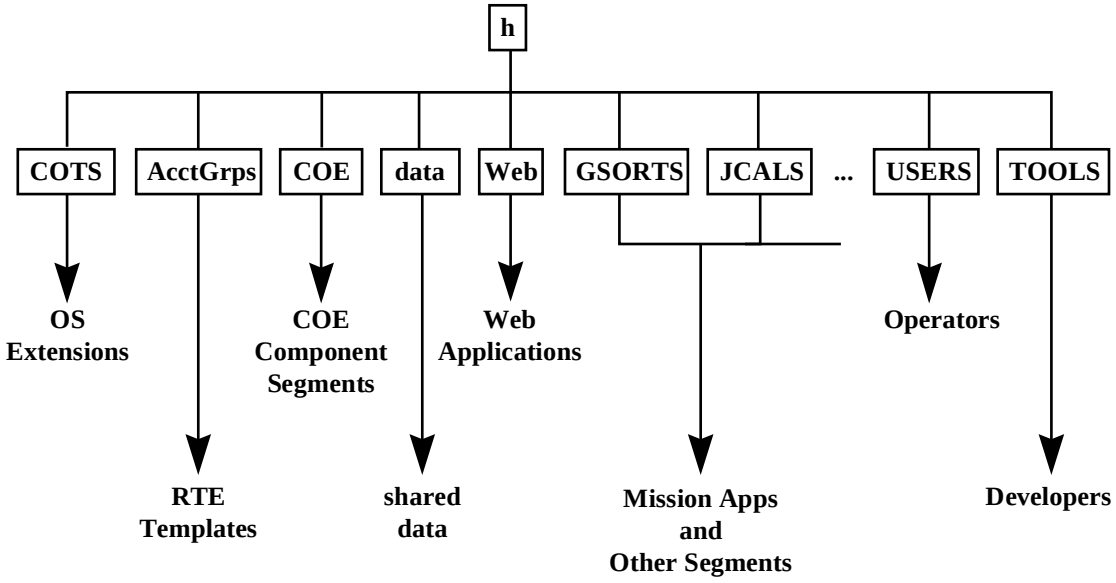


Figure 5-1: DII COE Directory Structure

Developers may *not* directly alter or create files outside of their assigned segment directory. DII compliance mandates strict adherence to this directive, with the following exceptions:

1. Temporary files may be placed in the operating system temporary[35] directory. For UNIX, this is the directory pointed to by `TMPDIR` (typically `/tmp`). For NT, use the applicable Windows API to locate the temporary directory. However, disk space is limited so developers must use this temporary directory sparingly and shall delete temporary files when an application is done.

2. Segments may place data files in the `/h/data` directory, and are required to do so for shared data (see subsection 5.4.4).

3. Operator-specific data files shall be placed in subdirectories underneath `/h/USERS` (see subsection 5.2.2).

4. Files may be added to the `/h/TOOLS` directory. This is a community directory for tools useful in the development process. Segments shall not place any files in this directory which are required at runtime since this directory is not installed at operational sites. This directory is described in subsection 5.2.3.

5. Segments may request that the COE tools modify community files during the installation process.

6. Segments may issue a request to modify a file to the segment which "owns" the file. This shall be done through use of, and only through use of, published APIs.

As software is loaded onto the system, the `/h` disk partition may eventually run out of disk space. The COE installation software will automatically create a symbolic link[36] to preserve the logical structure shown in

[35] For UNIX, the COE deletes all files in the temporary directory when the system is rebooted. This does *not* occur for NT system. Developers should make it a habit to delete all temporary files when they are finished and not rely upon the operating environment to delete them. This will ease porting problems and is a matter of good programming practice.

[36] Symbolic links are called *shortcuts* in NT. They are not identical concepts but are sufficiently similar for this discussion.

Figure 5-1, and delete the link when segments are removed. Hence, Figure 5-1 represents a *logical* view, not a *physical* view, of file and directory locations. Due to the potential need to relocate segments at installation time based on available disk space, DII-compliant segments must meet the following requirements:

- Segments shall use relative pathnames instead of absolute pathnames.
- Segments which use symbolic links to point to files contained within the segment shall use relative pathnames for the link.
- Segments which use symbolic links to community files may use absolute pathnames as long as (a) the segment can determine the community file's location at install time and (b) the segment can resolve linking to a community file which may itself be a symbolic link.
- (UNIX) Segments which add an environment variable to the account group's global runtime environment for locating files within the segment shall use a single "home" environment variable. Environment variables of this nature are normally required only when the segment files are to be accessible by other segments. Addition of the "home" environment variable is done by the segment installer through use of extension files and must *not* be done directly by the segment.

To illustrate the last requirement, consider a segment that provides a continuous readout of time-until-impact for a missile. Assume the segment's assigned directory is `MissleTDA` and it's segment prefix is `MSLE`. The `ReqrdScripts` segment descriptor (see subsection 5.5.2.22) is used to add the following to the account group's `.cshrc` file:

```
setenv MSLE_HOME  /h/MissleTDA
```

`MSLE_HOME` is called the segment's *home environment variable*. Static data within the segment can be referenced by `$MSLE_HOME/data` while executables may be referenced by `$MSLE_HOME/bin`. This technique of using relative pathnames means that segments can be easily relocated at development, integration, or installation time by modifying a single environment variable.

The last requirement stated above does not apply to environment variables defined for use purely within the software development environment. The COE requires that the runtime environment be separated from the development environment. This is typically done by separating environment variables and other settings into physically separate files. The development environment is not present during runtime for the operational system.

Also carefully note that the last requirement stated above applies only to the account group's *global* runtime environment, not a *local* runtime environment. When a segment executable is launched, it inherits the environment established by the account group template. It may then add to its local runtime environment through techniques equivalent to the C `putenv()` function.

The time-to-impact example illustrates additional COE requirements regarding definition of a home environment variable.

- A segment home environment variable shall point to the segment's assigned directory, *not* a lower level subdirectory (e.g., point to the directory `/h/MissleTDA` and *not* to the directory `/h/MissleTDA/Scripts`).
- (UNIX) A segment home environment variable, if added to the global environment, shall be added through an environment extension file (see `ReqrdScripts`).
- If a segment home environment variable is required, it shall be named *`segprefix`*`_HOME`, where *segprefix* is the segment prefix. Segments which use the same segment prefix must ensure that only

one segment defines a home environment variable. This requirement assures that home environment variables are uniquely named between segments.

- Segments shall not define a global environment variable that can be derived from an already-defined environment variable. For example,

```
setenv MSL_DATA          $MSL_HOME/data
```

is redundant and is therefore not allowed because the expression `$MSL_HOME/data` can be used wherever `$MSL_DATA` can be used.

- Segments shall not use the "~" character (or NT equivalent) to specify relative pathnames in the runtime environment, whether to define a home environment variable or any other environment variable.

UNIX allows statements of the form

```
source ~/Scripts/.cshrc.tst
```

in `.cshrc`, `.login`, and similar scripts. The "~" character is substituted at run time with the name of the home login directory (as defined in the `/etc/passwd` file). Suppose this statement were contained in a `.cshrc` file and, to prevent making duplicate copies and managing updates to this file, another segment wishes to use the UNIX `source` command to include this `.cshrc` file in its own environment. Any segment wishing to source the example `.cshrc` file must duplicate the same disk directory path structure (e.g., must have a `Scripts` subdirectory underneath the home login directory) and must have a file called `.cshrc.tst` underneath the `Scripts` subdirectory. This approach is problematic in the runtime environment because the login home directory is different for every operator, and leads to difficulties in sharing environment settings.

> **Note:** Developers should minimize the use of environment variables whenever possible. The amount of memory the operating system makes available to store environment variables is limited and is therefore a scare system resource. Also, developers should bear in mind that environment variables with shorter names require less memory to store than environment variables with longer names.

5.2.1 Segment Subdirectories

DII compliance mandates specific subdirectories and files underneath a segment directory. These are shown in Figure 5-2 for a general segment. The precise subdirectories and files required depend upon the segment type. For example, a `Scripts` subdirectory is required for account group segments. The `Scripts` subdirectory on a UNIX system will normally contain, as a minimum, `.cshrc` and `.login` scripts. These serve as a template for establishing a basic runtime environment. For software segments, the `Scripts` subdirectory contains environment extension files.

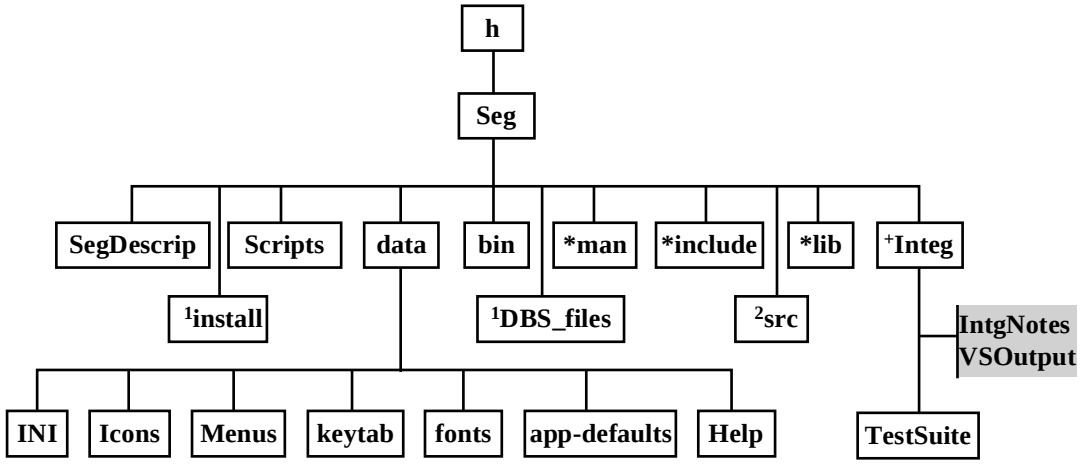
Some of the subdirectories shown in Figure 5-2 are required only for segment submission and are not delivered to an operational site. Runtime subdirectories normally required are as follows:

`data` subdirectory for static data items, such as menu items or help files, that are unique to the segment but will be the same for all users on all platforms

`bin` executable programs and shared libraries for the segment

Scripts directory containing script files (This is usually not required for NT platforms but, if required, the directory contains "batch" files.)

SegDescrip directory containing segment descriptor files.



*** Required for segments with published APIs**
+ Required for segment submission
1 For Database segments only
2 Recommended location for source code during development, Required location for source code delivered to DISA.

Figure 5-2: Segment Directory Structure

The descriptor directory `SegDescrip` is *always* required for *every* segment. Its contents are defined in later subsections. Segment developers may use arbitrary disk file structures during the development phase, but segments shall conform to the structure shown prior to submitting a segment to DISA. It is a violation of the COE to use a different subdirectory name to fulfill the same purpose as any subdirectory shown as a required subdirectory, or to use a different runtime directory structure than that shown in Figure 5-2.

For example, the subdirectory `src` is a recommended directory for the location of source code during software development. Developers are free to use this name, or any other structure convenient for their development practices. They *must*, however, use this directory name for source code delivered to the DISA SSA. `bin` is a required subdirectory and shall not be used for any purpose other than that stated in the *I&RTS*.

The distinction between the `Scripts` subdirectory and the `bin` subdirectory is subtle. Files in the `Scripts` subdirectory are used to establish attributes of the runtime environment. Scripts are used here in the sense of traditional UNIX, X Windows, or Motif files (`.cshrc`, `.login`, etc.) that are usually referred to only during the login process or in the establishment of a separate runtime session. Files of this nature are located in the `Scripts` subdirectory. Executable files may be created as a result of compiling a program or may be written as a shell. Files of this nature implement executable features of the segment and are located in the `bin` subdirectory.

Subdirectories `install` and `DBS_files` are only used for database segments. Their use is described in subsection 5.4.5

Subdirectories underneath `data` depend upon whether or not the segment has menu or icon files, uses DCE (subdirectory `keytab`), is NT-based and uses initialization files (subdirectory `INI`), or needs additional fonts or app-defaults. During segment installation (for UNIX platforms) special processing is performed on files within the `app-defaults` and `fonts` subdirectories. See subsection 5.4.4 for more details. See Chapter 6 for information on using "`.ini`" files on NT platforms.

The remaining subdirectories shown in Figure 5-2, except for `src`, are required in order to submit a segment to DISA as follows:

`include` subdirectory containing C/C++ header files or Ada package definition files for public APIs

`lib` subdirectory containing object code libraries for public APIs

`man` subdirectory containing UNIX "man" pages for public APIs

`Integ` subdirectory containing items required in the integration process

Segments which do not contain public APIs need not submit `include`, `lib`, or `man` subdirectories. For those segments with public APIs, private APIs are not allowed in the `include` subdirectory, nor are private libraries allowed in the `lib` subdirectory.

The `Integ` subdirectory serves as a convenient repository for information that needs to be communicated from the developer to the integrator. The file `VSOutput` is *required* for all segments submitted. The subdirectory `TestSuite` is *required* for all segments which submit public APIs and is to contain source code for a program(s) which exercises all APIs submitted. The file `IntgNotes` is *required* for all segments submitted and contains a brief description of why the segment is being submitted (new features, bug fixes, etc.). It also contains any special instructions that need to be communicated to the integrator for proper segment integration and installation.

5.2.2 USERS Subdirectories

The COE establishes individual operator login accounts and provides a separate subdirectory on the disk for storing operator-specific data items. The structure underneath this directory is created and managed automatically as accounts are added and deleted by the Security Administrator software. Developers who require access to any file maintained here (last profile selected, location of operator preferences files, etc.) shall use COE-provided APIs to access them and not rely upon a particular directory or file structure.

All users with valid accounts will have a subdirectory underneath `/h/USERS`. The subdirectory name will have the same name as the login account name. As shown in Figure 5-3, operator accounts may be global or local in scope. A *local* account is platform-specific, whereas *global* accounts are available from any platform on the LAN.

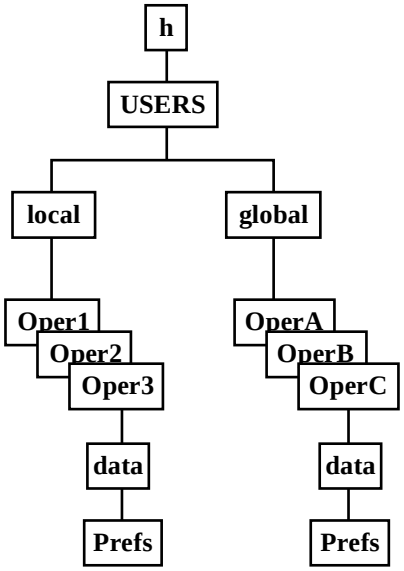


Figure 5-3: Operator Directory Structure

The subdirectory `Prefs` underneath the operator's data directory is used to store segment-specific operator preferences. DII compliance requires that segments store all operator preference data here. A segment is responsible for creating its own subdirectory (with the same name as the segment's assigned directory) and any required files when the segment first references the preferences data. The exact pathname for the `Prefs` subdirectory will change each time a different operator logs in, thus segment software shall use functions from the *Preferences Toolkit* APIs to retrieve the correct pathname for the currently active operator account.

Account group segments define the environment variables `USER_HOME` and `USER_DATA` to point to the correct operator directories. For the example in Figure 5-3, the following assignments would be made when the user whose login account name is `OperA` logs in:

```
USER_HOME = /h/USERS/global/OperA
USER_DATA = /h/USERS/global/OperA/data
```

Note that `USER_HOME` is *not* defined to be `/h/USERS/global/OperA/Scripts` which is the login home directory.

Segments, such as the Executive Manager, may need to reference menu and icon files for the operator's currently-defined profile. However, the directory location for these files is profile-dependent and will change during a login session if the operator changes profiles. Segments must use functions contained in the *Preferences Toolkit* APIs to determine the current profile. The environment variable `USER_PROFILE` is set by the account group segment during login, but segments must use APIs from the *Preferences Toolkit* to access files or directories related to individual operators, or to determine the current user profile.

DII compliance requires adherence to the following:

- Segments shall create subdirectories as needed under the operator's `Prefs` subdirectory for storing operator-specific data.

- Segments must work in an environment in which accounts are created and deleted. This requires that a segment create and initialize missing operator-specific data files.

- Account group segments shall set the environment variables `USER_HOME`, `USER_DATA`, and `USER_PROFILE`. (See footnote below. Account groups must still set `USER_PROFILE` in the interim to support legacy usage.) No other segment shall set or alter these environment variables.

- Segments shall determine the operator's directory and profile exclusively through the *Preferences Toolkit* APIs or the environment variables `USER_HOME`, `USER_DATA`, and `USER_PROFILE`.[37]

5.2.3 Developer Subdirectories

Software for the runtime environment is obtained by loading the desired mission-application segments and the required COE components. But the development environment is provided separately as a Developer's Toolkit because it is not delivered to, nor required at, an operational site. The Developer's Toolkit includes object code libraries, header files which define the public APIs, and various tools. By convention, tools are loaded underneath the `/h/TOOLS` subdirectory shown in Figure 5-1. This serves as a convenient directory for software contributed by the community for general development use.

5.2.4 Test Installation Subdirectories

The COE provides the ability for sites to temporarily install a segment on a platform to test it before putting it on other platforms on the LAN. This is accomplished by the `COETestInstall` tool, while removal of the test segment is accomplished by the `COETestRemove` tool (see Appendix C). These tools create temporary directories for storing the test segment and, if the segment already exists, `COETestInstall` moves the old segment to a safe location so that it can be restored by `COETestRemove` once the test is completed. Developers do not need to do anything special to their segment to enable this capability. It is handled automatically by the tools.

5.2.5 Application-Server Subdirectories

To assist site administrators, the COE provides support for creating application servers.[38] This is done by the tools `COECreateAS`, `COEConnectAS`, and `COERemoveAS` (see Appendix C). The `COECreateAS` tool allows segments to be loaded onto a platform that is to be configured as an application server. The application server may contain segments for mixed hardware types (e.g., Hewlett Packard [HP], Solaris, DEC, International Business Machines [IBM]). Figure 5-4 shows the directory structure maintained on the application server.

The tool `COERemoveAS` removes segments from an application sever. The tool `COEConnectAS` connects a client platform to an application sever. It also allows "dynamic" loading of segments as explained in Appendix C.

The COE does *not* support installation of multiple versions on the application server, for the same platform and operating system version. This could otherwise lead to problems if two different versions of a segment for the same platform type were executed at the same time. Temporary testing of a new segment version must be performed using the `COETestInstall` and `COETestRemove` tools described in subsection 5.2.4

Developers do not need to do anything special to their segments to enable the application-server capability. It is handled automatically by the tools.

---

[37] `USER_PROFILE` is preserved for backwards compatibility only. The COE allows there to be multiple active profiles so that an environment variable may not be the most appropriate way to determine the current user profile. Developers must not directly access this environment variable because its use may be phased out in a future release.

[38] Application servers are supported for UNIX platforms *only* in this *I&RTS* release.

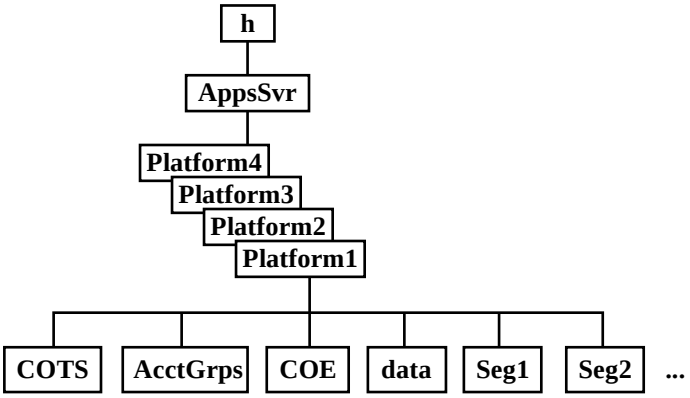


Figure 5-4: Applications Server

5.3 Segment Prefixes and Reserved Symbols

Each segment is assigned a unique subdirectory underneath `/h` called the *segment's assigned directory*. The assigned directory serves to uniquely identify each segment, but it is too cumbersome for use in naming public symbols. Therefore, each segment is also assigned a 1-6 character alphanumeric string called the *segment prefix*. The segment prefix is used for naming environment variables and things such as public APIs and public libraries where naming conflicts with other segments must be avoided. All segments shall prefix their environment variables with *`segprefix`*`_` where *segprefix* is the segment's assigned prefix. For example, the Security Administrator account group segment is assigned the segment prefix `SSO`. All environment variables for this segment are therefore prefixed with the string "`SSO_`".

The segment prefix is also used to uniquely name executables and shared libraries. All COE-component segments shall use the segment prefix to name executables and it is strongly recommended that all segments follow the same convention. For example, a proper executable for the Security Administrator account group is `SSOSetClassif`. A properly named shared library would be `SSOSampleLib.lib`. This approach simplifies the task of determining the files that go with each segment and reduces the probability of naming conflicts.

> **Note:** Use the segment prefix inside application code in situations where it is important to distinguish one segment from another. For example, when audit information is written to the security audit log, the segment prefix is also written to the audit log to allow determination of which application module generated the audited event. The same advice applies to all audit logs, including those maintained by the operating system or a DBMS.

It is sometimes convenient for segments to share the same segment prefix. This is true for aggregate segments or for segments produced by the same contractor. The COE allows segments to share the same segment prefix; however, the burden for avoiding naming conflicts is placed on the segment developer.

> **Note:** This means that segment prefixes are not guaranteed to be unique and therefore cannot be used to uniquely identify a segment. Each segment shall have a uniquely assigned directory and segment name. Therefore, the name or directory in combination can be used to uniquely identify a segment. There are situations where it is more convenient to specify a segment's assigned directory rather than its name, such as in `COEFindSeg`, because the directory name is typically shorter than the segment name and this fact can be useful in speeding up character string comparisons in segment searches. Furthermore, because the segment directory will not have embedded blanks but the segment name may, the segment name will not necessarily be the same as the assigned directory name.

The segment prefixes shown in Table 5-1 are reserved.

Segment Prefix	Applicability
CBIF	Character-Based I/F account group segment
CDE	Common Desktop Environment segment
COE	Common Operating Environment segment
DBA	Database Administrator account group segment
DCE	Distributed computing environment segment
DII	Defense Information Infrastructure segment
ECEDI	Electronic Commerce/Electronic Data Interchange segment
ECPN	Electronic Commerce Processing Node segment
EM	Executive Manager segment
GCCS	Global Command and Control System segment
GCSS	Global Command Support System segment
INFRMX	Informix COTS segment
JCALS	Joint Computer-Aided Acquisition and Logistics Support segment
JMCIS	Joint Maritime Command Information System segment
JMTK	Joint Mapping Toolkit segment
MOTIF	Motif
NIPS	Navy NIPS segment
NT	Generic NT segment
ORACLE	Oracle COTS segment
OSS	Navy OSS segment
SA	System Administrator account group segment
SCO	SCO-UNIX segment
SSO	Security Administrator account group segment
SYBASE	Sybase COTS segment
TIMS	Navy TIMS segment
UB	Navy Unified Build segment
UNIX	UNIX operating system
USER	prefix for operator-specific items
WIN	generic Windows segment
WIN95	Windows 95 segment
WINNT	Windows NT segment for 80x86 platforms
XWIN	X Windows

Table 5-1: Reserved Segment Prefixes

The COE sets five environment variables that must not be confused with the USER prefix or the segment home environment variable.

- The HOME environment variable is set by the operating system to be the login directory; that is, the login directory as contained in the UNIX /etc/passwd file. This will normally point to a Scripts subdirectory while the segment "home" environment variable (*segprefix*_HOME) is one level up from HOME.

- The USER environment variable is set by the operating system to be the login account name and does not refer to a directory as does the USER prefix. Thus, USER_HOME will be /h/USERS/$USER.

- The environment variables LOG_NAME, LOGNAME, and LOGIN_NAME are equivalent to the USER environment variable[39], but are not always present on every system.

The COE also includes a number of predefined environment variables that are required by UNIX, NT, X Windows, and other COTS software. These environment variables are either set automatically by the operating system or they must be set by an account group segment. Other segments shall not alter these environment variables except as permitted by environment extension files (e.g., extending the path environment variable).

Table 5-2 lists various important environment variables that are set by the applicable account group, the parent COE-component segment, or the COE installation tools.

The COE sets environment variables MACHINE_CPU and MACHINE_OS to define the hardware and operating system being used. This allows scripts and descriptors to perform operations that are dependent on the hardware or operating system. Table 5-3[40] lists the possible values set by the COE which either may be used as constants in #ifdef constructs within descriptor files or as possible values for the appropriate environment variable (e.g., MACHINE_CPU).

Note that the environment variables (e.g., MACHINE_CPU) will have one and only one value, but several constants may be defined for use within the descriptor files. For example, if the hardware platform is an HP715 running HP-UX 9.01, the MACHINE_CPU environment variable will be set to HP715, MACHINE_OS will be set to HPUX, while the constants HP, HP715, HPUX will be defined for use in descriptors.

[39] USER is preserved for backwards compatibility with legacy pre-POSIX systems. LOGNAME is the proper POSIX equivalent.

[40] This list of constants will be updated as new platforms are supported. Refer to the *DII COE Release Notes* and *Version Description* documents for details as new platforms are supported.

Environment Variable	Usage
COE_SYS_NAME	string containing system name (e.g., "GCCS")
+ COE_TMPSPACE	location of temporary space
* DATA_DIR	/h/data
DISPLAY	current display surface (UNIX only)
HOME	user's login directory
+ INSTALL_DIR	absolute pathname to where segment was installed
* LD_LIBRARY_PATH	default location of shared X and Motif libraries (UNIX only)
* LOGNAME	user's login account name
* LOG_NAME	user's login account name
* LOGIN_NAME	user's login account name
* MACHINE_CPU	CPU type derived from uname -m
* MACHINE_OS	Operating system derived from uname -s -r
path	list of paths to search to find an executable
SHELL	shell used (e.g., /bin/csh) (UNIX only)
+ SYSTEM_ROOT	absolute pathname to where Windows is installed (applicable to PC-based COE only)
TERM	terminal type (UNIX only)
* TMPDIR	location of the system-defined temporary directory
* TZ	time zone information (UNIX only)
USER	user's login account name
USER_DATA	user's data directory under /h/USERS/local or /h/USERS/global
USER_HOME	user's home directory under /h/USERS/local or /h/USERS/global
USER_PROFILE	user's current profile under /h/USERS/local/Profiles or /h/USERS/global/Profiles
* XAPPLRESDIR	/h/data/app-defaults (UNIX only)
* XENVIRONMENT	/h/data/app-defaults/COEBaseEnv (UNIX only)
* XFONTSDIR	/h/data/fonts (UNIX only)

Legend:
* Environment variables set by the parent COE-component segment.
+ Environment variables set by the COE installation tools. These are defined only at installation time.
All remaining environment variables are set by the applicable account group segment.

Table 5-2: COE-Related Environment Variables

MACHINE_CPU Environment Variable	
Constant	**Platforms for Which Defined**
DEC	DEC Alpha platforms
HP700	HP 700 series platforms
HP712	HP712 platforms
HP715	HP 715 platforms
HP750	HP 750 platforms
HP755	HP 755 platforms
IBM	IBM RISC 6000 platforms and PowerPC
PC386	Intel 80386 platforms
PC486	Intel 80486 platforms
PENTIUM	Intel Pentium platforms
SGI	Silicon Graphics platforms
SPARC	Sun Sparc platforms
SUN4	Sun 4 platforms

MACHINE_OS Environment Variable	
Constant	**Platforms for Which Defined**
AIX	IBM RISC 6000 platforms and PowerPC
OSF1	DEC Alpha platforms
HPUX	all HP-UX platforms
IRIX	Silicon Graphics platforms
NT	all NT platforms
SOL	all Solaris platforms
WIN95	all Windows 95 platforms

Miscellaneous Constants	
Constant	**Platform for Which Defined**
DEC	all DEC platforms, regardless of OS
HP	all HP platforms, regardless of OS
IBM	all IBM platforms, regardless of OS
PC	all 80x86 platforms, regardless of OS
SGI	all SGI platforms, regardless of OS
SPARC	all Sun Sparc platforms, regardless of OS

Table 5-3: Platform and Operating System Constants

5.4 Segment Types and Attributes

Segment types and attributes were briefly introduced at the beginning of this chapter. The present subsection describes segment types and attributes in more detail. Segments are the cornerstone of the COE approach, and proper determination of their type and associated attributes determines how the COE handles them. Developers have considerable freedom in building segments; however, there are some important considerations regarding them.

- Creation of an account group segment requires prior approval by the Chief Engineer. Most account groups are predefined by the COE itself to establish DII-compliant runtime environments. System designers will typically add an operator account group that establishes the basic runtime environment for their system. Other developers will not normally create account group segments.

- Creation of a COE-component segment requires prior approval by the DII COE Chief Engineer.

- All COTS products shall be packaged as individual COTS segments, unless approved by the DII COE Chief Engineer. This requirement is mandated to make it easier to handle COTS licenses, and to ensure that a single version of a COTS product is in use. Dependencies on COTS product versions must be identified and coordinated with DISA to ensure that the proper version is supported by the COE.

- Segments shall not modify any file that lies outside the segment's directory. Community files may be modified only through public APIs or through requests made to the COE installation tools.

Segment types are identified by the `$TYPE` keyword in the `SegName` descriptor. Segment attributes are also specified in the `$TYPE` keyword by the presence of an optional attribute parameter. See subsection 5.5.1.10 for details.

5.4.1 COTS Segment Types

The COTS segment type is used to describe the installation of COTS products. It is preferable to structure a COTS product as a software segment, if at all possible, because it provides more control over the installation and placement of the COTS product. However, this is sometimes not possible because where COTS products will be loaded, what environment extensions are required, etc. are often very vendor-specific.

The COE must retain segment information about all segments, including COTS products. The segment descriptor information for all COTS segments is located underneath the directory `/h/COTS` as shown in Figure 5-5. COTS software is not necessarily actually stored in the directory `/h/COTS`. Frequently only the segment descriptor information is stored there because the actual location of COTS products is often spread across several subdirectories (such as `/usr`, `/usr/lib/X11`, and `/etc`).

Using UNIX as the example, Figure 5-5 shows the segment descriptor information for the operating system (`UNIX`), the X Windows environment (`XWindows`), the Motif window manager and libraries (`Motif`), and the Common Desktop Environment software (`CDE`). These four subdirectories, along with the actual COTS software, are loaded with the COE kernel. The example in Figure 5-5 also shows that the DCE COTS product has been installed.

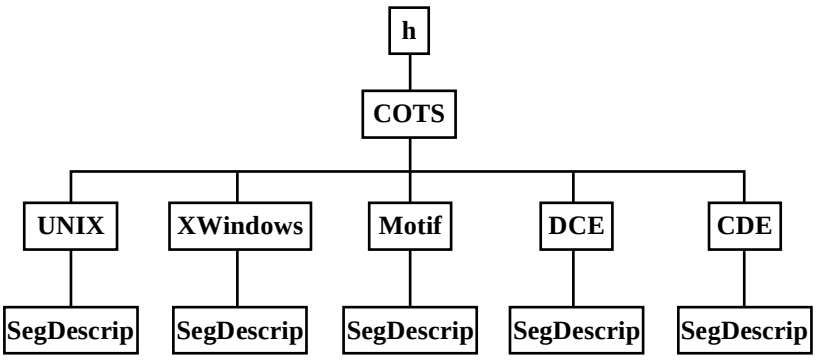


Figure 5-5: COTS Directory Structure

COTS products sometimes have very specific requirements as to the location of files within the product. The general approach to such segments is to create a temporary segment structure in which to store the COTS product, copy the COTS files to their required location during installation, and then copy the segment descriptor information to `/h/COTS`. It is the responsibility of the `PostInstall` script to copy the COTS files to their appropriate directories and to perform any other required initialization steps. The installation software handles moving the segment descriptor information to the standard location, `/h/COTS`.

For example, assume a COTS product called `SampleCots` is to be installed which requires loading a series of files into `/etc` (files `f1`, `f2`, and `f3`), `/usr/local` (files `f4` and `f5`), and `/usr/lib` (files `f6`, `f7`, `f8`, and `f9`). A segment directory structure can be set up in whatever manner is most convenient. Figure 5-6 shows one possible solution. The installation software will load the segment `SampleCots` wherever there is room on the disk and will set the environment variable `INSTALL_DIR` to the absolute pathname to where `SampleCots` was loaded. The `PostInstall` script for this example must recursively copy the subdirectories `etc` and `usr` from `INSTALL_DIR` to `/etc` and `/usr`. The installation software will copy the segment descriptor information to `/h/COTS/SampleCots` and then delete all files underneath `INSTALL_DIR`.

As an alternative, the COE allows a segment to specify exactly where it must be loaded. This is done with the `$HOME_DIR` directive described in subsection 5.5. This reduces the need to copy files from one directory to another, and eliminates the temporary disk space required during installation (e.g., to temporarily store the segment when it is read from tape, then copy it to its new location, then delete the temporary location).

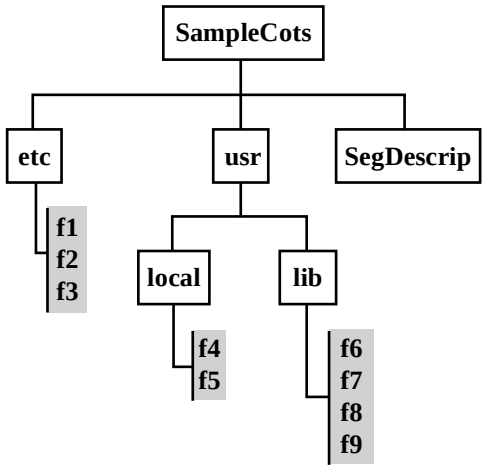


Figure 5-6: Example COTS Segment Structure

The segment descriptor `FilesList` (see subsection 5.5.2.13) is used to document where a COTS product was installed. The `FilesList` descriptor for this example is

```
$PATH:/etc
$FILES
f1
f2
f3
$PATH:/usr
$FILES
local/f4
local/f5
lib/f6
lib/f7
lib/f8
lib/f9
```

To summarize the COTS segment type:

- COTS products should be installed as a software segment type if possible.
- The COTS segment's `PostInstall` script is responsible for copying files to their required location. The `PostInstall` script must ensure that enough space exists.
- The installation software places the segment descriptor information underneath `/h/COTS/`*`SegDir`* where *SegDir* is the segment directory name chosen for the temporary segment structure (`SampleCots` in the example above).
- The COTS segment's `PostInstall` is responsible for deleting the temporary segment structure after the installation is complete.
- COTS segments shall document what files are loaded and their location in the `FilesList` segment descriptor.
- When practical, COTS segments should make symbolic links to the appropriate location for their software instead of copying the files and directories. This allows the installation software to make more effective use of the disk space available and avoids the problem of running out of disk space for such common directories as `/usr` and `/etc`.

Note: Developers should normally not include the vendor name in the segment name because this makes the segment vendor-specific. Other segments which then depend upon the COTS product are affected because they then become vendor-specific as well. For example, a segment name such as "DCE" is preferable to "Vendor A DCE" because segments may specify a dependency on a segment whose name is "DCE" rather than "Vendor A DCE." This is especially the case when the COTS product is the implementation of an industry standard. However, it is sometimes advisable to include the vendor name because the

product truly is vendor-proprietary. This is typically the case with an RDBMS.

5.4.2 Account Group Segment Types

An account group segment is a template for establishing a basic runtime environment context that other segments may extend in a controlled fashion. An account group segment determines

- the processes to launch,
- the order in which to launch processes, and
- the required environment script files (`.cshrc`, `.login`, etc.).

Account groups may also contain executables and data in the subdirectories identified in Figure 5-2.

The COE provides several predefined account groups. They are located underneath `/h/AcctGrps` shown in Figure 5-1. Important predefined account groups include the following:

`CharIF`	account group for character-based interfaces
`DBAdm`	account group for database administrators
`SecAdm`	account group for security administrators
`SysAdm`	account group for system administrators

In addition to these account groups, COE-based system designers will generally create their own account group for normal operator accounts (`GCCS` for the Global Command and Control System, `GCSS` for the Global Command Support System, `ECPN` for the Electronic Commerce Processing Node system, etc.). They will include `CharIF` if the system supports a character-based interface and may include other account groups to suit system mission requirements.

Figure 5-7 shows how the UNIX System Administrator account group is structured. It demonstrates what account groups are for and how they are used in building a COE-based system.

`bin` Subdirectory

Account groups utilize COE executables, located underneath `/h/COE/bin`, but will usually include additional account group specific programs. These are located in the account group's `bin` subdirectory. DII compliance requires that executables within this subdirectory use the segment prefix to avoid potential naming conflicts with other executables.

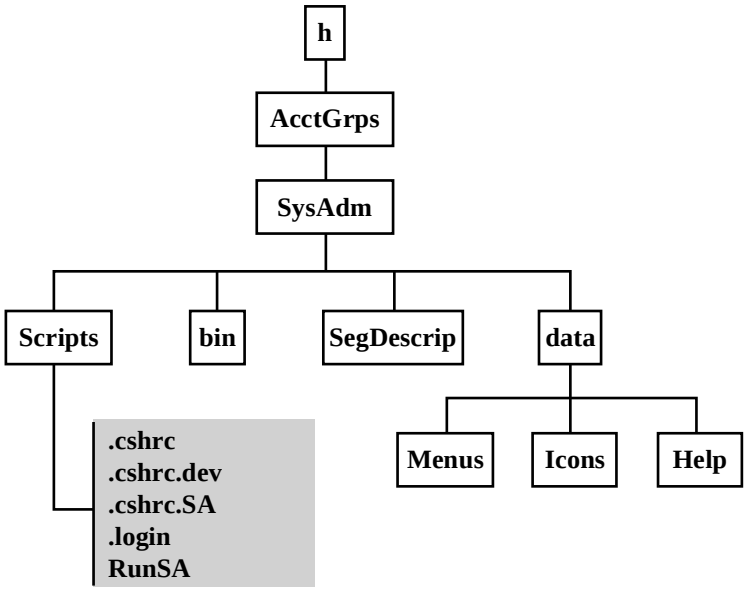


Figure 5-7: Example Account Group Directory Structure

`data` Subdirectory

Segment data specific to the System Administrator account group is located in the `data` subdirectory. The `Menus` subdirectory contains menu files that have menu entries for all options available from the basic System Administrator application. The segment installation software may modify files contained here to add other menu options. Account group menu files are used as templates from which profiles are created by including or excluding desired menu items and execution permissions. The `Icons` subdirectory is analogous, but defines icons for use by the desktop for launching applications.

Help files are located underneath the `data/Help` subdirectory and identified through the `Help` segment descriptor. Refer to subsection 5.5.2.15 for more details on this segment descriptor.

`Scripts` Subdirectory

A UNIX account group segment will usually contain at least the following two scripts to establish the runtime environment:

`.cshrc`	define environment variables
`.login`	define terminal characteristics

Precise contents of these files is application-dependent. Other segments may be loaded to extend the environment established by the account group. This is done through *environment extension files*. DII-compliant account group segments shall name environment extension files in the form

`scriptname.segprefix`

where *scriptname* is the environment file to be extended and *segprefix* is the segment prefix. For the example shown in Figure 5-7, the environment extension files are:

`.cshrc.SA`	extensions to the `.cshrc` file
`.login.SA`	extensions to the `.login` file

Extension of the `.login` file is seldom required.

Environment extension files permit COE installation software to provide segment-specific environment modifications. A segment uses the descriptor `ReqrdScripts` (see subsection 5.5.2.22) to indicate which environment file to extend and the installation tools modify the proper file within the account group segment.

For example, suppose the installation tools have loaded a segment underneath `/h/SAOpt` and the `SAOpt` segment has an environment extension file named `.cshrc.SAOpt` in the segment's `Scripts` subdirectory. The installation tools will include the new environment settings by inserting the following statements in the account group's file `.cshrc.SA`:

```
if (-e /h/SAOpt/Scripts/.cshrc.SAOpt) then
      source /h/SAOpt/Scripts/.cshrc.SAOpt
endif
```

The installation tools automatically remove these statements from `.cshrc.SA` if the segment `SAOpt` is deleted.

Account group segment developers shall ensure that environment extension files are included and accounted for in the appropriate account group segment's scripts. For example, the `.cshrc` file shown in Figure 5-7 includes the following statements

```
if (-e $SA_HOME/Scripts/.cshrc.SA) then
      source $SA_HOME/Scripts/.cshrc.SA
endif
```

to account for `.cshrc` extensions. Also note that the `source` command shall be of the form

```
source $SA_HOME/Scripts/.cshrc.SA
```

rather than

```
source $USER_HOME/Scripts/.cshrc.SA
```

The COE-mandated form ensures a single copy of the environment extension file, updated and maintained by the installation software.

The file `.cshrc.dev` shown in Figure 5-7 relates to the software development environment. It is not a required file, but is described here as an example of how the development environment can be accommodated, yet kept separate from the runtime environment. In the example shown, developer preferences such as alias commands are included in `.cshrc.dev`. These preferences *must* not be included as part of the runtime environment. A technique such as

```
if ($?DEVELOPER) then
      source $SA_HOME/Scripts/.cshrc.dev
endif
```

within the `.cshrc` file is required to achieve separation of the development environment from the runtime environment. This technique will not work for certain files, such as `.mwmrc`, because they do not support conditional statements.

Account groups must include the base environment established by the COE. Subsection 5.4.8 describes the COE-component segments in more detail. The `.cshrc` file in Figure 5-7 includes the base COE environment with the statements

```
if (-e /h/COE/Scripts/.cshrc.COE) then
      source /h/COE/Scripts/.cshrc.COE
endif
```

The remaining files in Figure 5-7 contain similar statements to include other COE environmental settings.

Account groups must also provide a script or program which launches the application. This is the file named `RunSA` shown in Figure 5-7. DII compliance requires this file to be located underneath the `Scripts` subdirectory.[41]

To summarize compliance requirements for account groups:

- Account group segments shall provide environment extension files of the form *`scriptpname.segprefix`*, where *scriptname* is the name of the script which sets the environment, and *segprefix* is the account group's segment prefix. This must be done for any files that other segments may extend (e.g., `.cshrc.SA` for the `SysAdm` account group).

- Account group executables shall use the segment prefix to avoid naming conflicts.

- Account group segments shall not include the developer environment as part of the runtime environment.

- Account group segments shall provide a single program or script with the name `Run`*`segprefix`*, where *segprefix* is the segment prefix, to initiate execution of the account group's application. This executable shall be located in the account group segment's `Scripts` subdirectory.

- Account group segments shall automatically include environment settings established in `/h/COE/Scripts`.

- Segment developers shall not modify account group files except through use of the installation software.

- Segment developers shall not override environmental settings established by the account group. Segments may use environment extension files to expand the environmental settings.

5.4.3 Software Segment Types

Software segments add functionality to one or more account groups. The account group(s) to which the software segment applies is called the *affected account group(s)*. The directory structure for a software segment was presented in Figure 5-2.

Software segments frequently need to extend the runtime environment, add new menus and icons to the desktop, and include new executables in the search path. Environment extension files are located underneath the software segment's `Scripts` subdirectory. The `ReqrdScripts` segment descriptor indicates which environment files are to be extended.

Software segments provide additional menu and icon files underneath the segment's `data/Menus` and `data/Icons` subdirectories respectively. The segment descriptors `Menus` and `Icons` are used to describe where the new items are to appear on the desktop. At installation time, the menu and icon files

[41] This program is required for backwards compatibility and as an aid to integrators and testers. It may be phased out in a future release because the program is not necessarily used in the operational system, depending upon the characteristics of the system desktop.

from all contributing segments are added to the affected account group. This then serves as a master template of all possible functions provided within the account group. Profiles are then created by selectively including or excluding functions within this master template.

UNIX segments that provide executables must ensure that the `bin` subdirectory is included in the search path. This is accomplished by including a statement of the following form in a `.cshrc` extension file:

```
set path =($path $segprefix_HOME/bin)
```

The segment shall append its `bin` subdirectory, and *only* its `bin` subdirectory, at the *end* of the search path, *not* the beginning. An implied aspect of this requirement is that segments cannot depend upon a specific loading sequence, other than that a segment will not be loaded until after all segments it depends upon are loaded. A specific requirement is that segments shall *not* insert the current working directory (i.e., ".") into the search path.

DII compliance requires the following:

- Segments shall not make separate copies of executables from other segments, the operating system, or other COTS products.
- Segments shall use environment extension files as necessary to extend the environment established by the affected account group.
- Segments shall use the segment prefix to name objects whenever conflicts may arise with other segments.
- Segments shall be completely self-contained. Dependencies on, or conflicts with, other segments shall be specified through the appropriate `Requires` or `Conflicts` segment descriptors.
- Segments shall not insert the current working directory into the search path for executables.
- (UNIX) Segments shall include their `bin` subdirectory at the end of the search path, not at the beginning nor in the middle.

5.4.4 Data Segment Types

Data files are most often created explicitly at runtime by a segment or loaded as part of the segment itself. However, the ability to load data as a separate segment is useful when there is classified data, optional data, large amounts of data, or data that may not be releasable to all communities. The COE supports five categories of data grouped according to data scope, how the data is accessed, and where the data is located:

Global — Data in this category means that every platform, every application, and every operator on the LAN accesses and uses exactly the same data. Global data is made available through Network File Server (NFS) mount points or some similar technique. Examples of global data include the track database and message logs. Global data is located in subdirectories underneath `/h/data/global`.

Database — This category is similar to global data but is used to provide data fill for a database segment. Examples of this kind of data include intelligence databases, JOPES data, and TPFDD data. Data is loaded into the appropriate objects previously created by a database segment in a database server. Database segments are discussed further in subsection 5.4.5. Data segments for databases are usually removed after successfully loading data into the database server.

Local Local data is limited in scope to an individual platform. All platform users and applications access the same data, but the data may (and frequently will) differ from one platform to another. Examples include overlays and briefing slides, although the COE provides techniques for exporting these to other platforms. Local data is located in subdirectories underneath `/h/data/local`.

Segment Segment data is local to a platform, but is managed and accessed by a single software segment. This data is located under the segment's `data` subdirectory (e.g., *`SegDir`*`/data` where *SegDir* is the assigned directory) and is typically static data used for segment initialization.

Operator Data in this category is specific to an operator and is the most limited in scope. Typical examples include preferences for map colors, location of various windows, and font size. Operator data is stored in a `data` subdirectory underneath `/h/USERS` created for the operator when the operator login account is created, as described in subsection 5.2.2.

There are some important considerations with respect to these data categories:

- Data is not necessarily available to an operator or process even if the data scope would otherwise permit it. Discretionary access controls limit access based upon the security policy of the system.

- In some cases, data that could be global is replicated on every platform to improve system performance. For example, World Vector Shoreline data is identical for everyone on the LAN, and hence meets the criteria for the global data category. However, for efficiency, this data may be replicated on each platform which requires maps and is thus considered local.

- Distinction is made between segment data and local data because it affects where the data is stored on the disk. Local data for all segments is stored in a single place to make it easier for doing data backups. Because segment data is normally static, it does not usually need to be archived and remains with the segment.

Segment data created at runtime or loaded as part of the segment does not require any special consideration by the COE. The remainder of this subsection will deal with the COE requirements for local and global data, and then present an example of how a data segment is structured for local, global, and segment scope.

`global` and `local` Subdirectories

Figure 5-8 shows the directory structure for global and local data. The COE runtime environment sets the environment variable `DATA_DIR` to point to `/h/data`. Segments shall use this environment variable to reference global or local data. The segment which owns the local or global data is responsible for creating and managing its data subdirectories underneath `$DATA_DIR/local` and `$DATA_DIR/global`. Assuming the segment's assigned directory is *SegDir*, the segment shall create a subdirectory of the form *`SegDir`*`/data` under `$DATA_DIR/local` and/or `$DATA_DIR/global` as appropriate.

For example, suppose a segment that does Anti-Submarine Warfare (ASW) planning is located underneath `/h/ASW` and it will create both global and local data. Then the ASW segment must create the subdirectory `$DATA_DIR/local/ASW/data` for local data and the subdirectory `$DATA_DIR/global/ASW/data` for global data.

The COE mandates that local and global data be structured in this fashion for the following reasons:

- Centralizing data makes it easier to archive and restore. A simple data archive/restore utility can be created without needing to know how many segments are loaded in the system.

- Separating data from software makes it simple to load the software without destructively overwriting existing data. This is especially important as segments are upgraded.

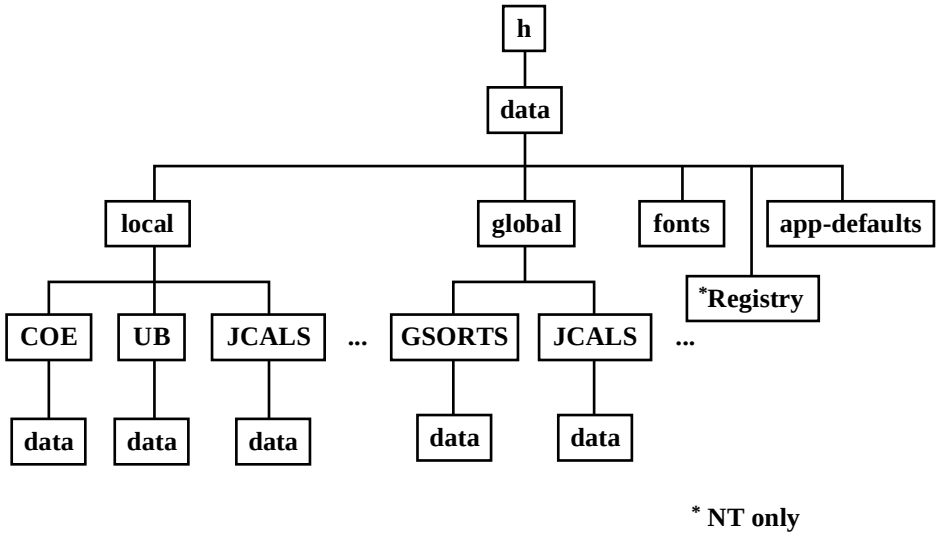


Figure 5-8: Data Directory Structure

- Collecting all global data under a single directory reduces the number of NFS-type mount points and improves overall network performance.

- Organizing data into a standard structure simplifies training and simplifies determination of what data is loaded in the system.

`fonts` and `app-defaults` Subdirectories (UNIX)

Figure 5-8 shows two additional subdirectories, `fonts` and `app-defaults`. These are applicable to UNIX only. The COE sets environment variables `XFONTSDIR` and `XAPPLRESDIR` to point to these subdirectories. Their purpose is to contain additional fonts (such as Naval Tactical Data System [NTDS] symbology) or application resource files that are not provided by the standard X/Motif distribution. It is a violation of the COE for a segment to overwrite or add files to the standard X/Motif distribution.

During installation, the installation tools look for subdirectories `data/fonts` and `data/app-defaults` underneath the segment's directory. Files contained within these subdirectories must use the segment prefix to guarantee unique names. The installation tools create symbolic links underneath the directory `$DATA_DIR/fonts` to every file in the segment's `data/fonts` subdirectory and removes the links when the segment is deinstalled. Similarly, links are created for files underneath the segment's `data/app-defaults` subdirectory.

Creating a data segment requires additional considerations. A segment structure is created for the data and the installation tools *logically* insert the data underneath `$DATA_DIR` for global and local scope, but underneath the parent segment for segment data. This is best described through use of an example.

Assume a mine countermeasures decision aid has an assigned directory of `MineTDA`. Assume that a separate data segment is to contain parametric data on floating, proximity, and land mines for the decision aid. Figure 5-9 shows the appropriate directory structure for the data segment. Further assume that when

installed, the decision aid is located underneath /h/MineTDA. Consider how the installation tools handle the mine data segment for global, local, and segment scope.

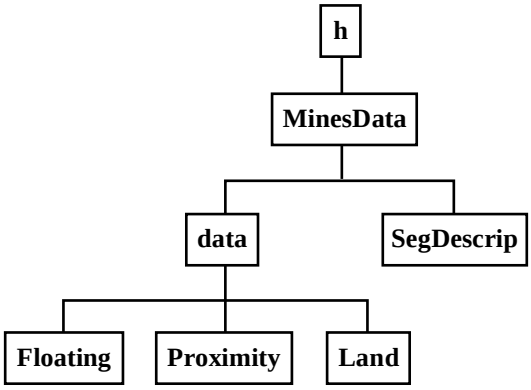


Figure 5-9: Example Data Segment Structure

Global Scope Example

The Data segment descriptor describes the data scope. For a global data segment, the installation tools will load the mine data underneath the directory $DATA_DIR/global/MinesData. If there is insufficient space to load the segment underneath $DATA_DIR/global, the install tools will report an error and abort. The mine TDA can thus reference global proximity-mine data as being underneath the directory $DATA_DIR/global/MinesData/data/Proximity.

Local Scope Example

For a local data segment, the installation tools will load the mine data on the first available disk partition. The installation tools will then create a symbolic link from $DATA_DIR/local/MinesData/data to wherever the data segment was actually loaded. That is, if the data segment is loaded underneath /home2/MineData, then the symbolic link will point to the directory /home2/MineData/data. The mine TDA can still reference local proximity mine data as being underneath the directory $DATA_DIR/local/MinesData/data/Proximity.

Segment Scope Example

For segment scope data, the installation tools will load the mine data on the first available disk partition. A symbolic link is then created from the directory /h/MineTDA/data/MinesData/data to wherever the data segment was actually loaded. Proximity data can thus be referenced as being underneath the directory $HOME_DIR/data/MinesData/data/Proximity.

It should now be clear why the COE requires that segments which dynamically create global or local data do so underneath a directory of the form *SegDir*/data, where *SegDir* is the name of the segment's assigned directory. This creates a uniform technique for locating files whether they are created directly by a segment or loaded as part of a data segment.

In summary, DII compliance mandates that:

- Segments shall create a data subdirectory underneath $DATA_DIR for global and local data if they own global or local data. The subdirectory created shall be *SegDir*/data where *SegDir* is the name of the segment's assigned directory.

- The parent COE-component segment shall set the environment variable `DATA_DIR` to point to `/h/data`.
- Segments shall use the environment variable `DATA_DIR` to reference data underneath `/h/data`.
- Segments are responsible for creating the segment's data subdirectories underneath `/h/data`.
- Segments are responsible for handling the case in which a data file is not present or is corrupted.
- (UNIX) The parent COE-component segment will set environment variables `XFONTSDIR` and `XAPPLRESDIR` to point to `$DATA_DIR/fonts` and `$DATA_DIR/app-defaults` respectively.
- (UNIX) Segments shall place fonts that need to be accessible via `XFONTSDIR` in the segment's *`SegDir`*`/data/fonts` subdirectory. Files in this subdirectory shall be named using the segment prefix.
- (UNIX) Segments shall place application resource files that need to be accessible via `XAPPLRESDIR` in the segment's `data/app-defaults` subdirectory. Files in this subdirectory shall be named using the segment prefix.

5.4.5 Database Segment Types

The database segment type is similar in concept to the data segment type, except that the data within a database segment type is managed by a DBMS. Data within a data segment type is typically organized as a "flat file" and is typically managed by the operating system's file system.

As explained in Chapter 2, a database segment has scope, which is an indication of how widely the data is shared, not of where the data is located, as is the case with the data segment type already described. This scope is indicated in the Database segment descriptor discussed in subsection 5.5.2.9. Data within a database segment type may be:

Unique — This type of database segment indicates that the data is used by only one application, or is under the configuration control of the segment sponsor. Unique data represents no sharing between segments.

Shared — This type of database segment indicates that the associated data is used by multiple mission-application segments or is managed across multiple database segments. Data is shared, but typically only within one mission domain (e.g., logistics, financial, command and control).

Universal — Data in this category represents the most extreme form of "shareability." These database segments represent widespread usage across mission domains, application segments, and require centralized configuration management.

A database segment contains everything that is to be installed on the database server under the management of the DBMS and the ownership of the DBA. It contains the scripts to create a component database and any utilities provided by the developers for the DBA's use in installing and filling that particular database. These scripts must include those for granting and revoking database roles. The only applications permitted in a database segment are those that support its installation and data fill or that extend DBMS services for the DBA. Database segments may only be installed on a database server.

When a database segment is installed it must first lay down any scripts, data files, etc. that will be used to create the database. These scripts are then executed by `PostInstall` to create the component database. They must first allocate storage to hold the database and create one or more database accounts to own that

database. They then can create the database within the storage just allocated and fill it with data. Finally, roles are created to manage access and the roles are given the appropriate privileges.

Developers cannot provide data files for the DBMS as part of the segment. Database files must be created using the DBMS vendor's utilities (e.g. Oracle's SQL*DBA `CREATE TABLESPACE` command) to be correctly incorporated in the DBMS instance.

Figure 5-10 is the same as Figure 5-2 except that it has been shaded to highlight the directories which are used only for database segments and directories which are not required at runtime have been removed. *Seg* is the segment's assigned directory. It is unique and, for a database segment, it must be the same as the name of the database owner account for the segment's data objects.

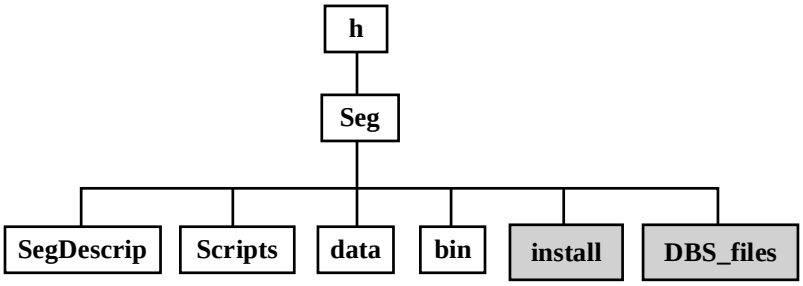


Figure 5-10: Database Segment Structure

`Scripts` Subdirectory

The `Scripts` subdirectory shall contain any segment-specific scripts needed to set the environment for the database installation. This includes environment variables for all directory paths that are used by the installation scripts. Note that this directory is used as a place to store installation-related environmental scripts. As with the development environment, scripts and environmental settings which are used only for installation must be kept separate from those used by the runtime environment.

`SegDescrip` Subdirectory

The `SegDescrip` subdirectory contains the descriptor files necessary to install the database segment. Certain information specific to database segments must be incorporated in the `SegInfo` file. The `Database` descriptor is used to identify information such as object dependencies that are within the database and therefore cannot be evaluated without the use of the DBMS. See subsection 5.5.2.9 for the associated keywords for this segment descriptor.

The `PreInstall` descriptor file should prompt the installer to provide the password for the DBMS' database administrator account. The password prompt must be implemented via the `COEPromptPasswd` API (see Appendix C) provided by the COE Services. The DBA password entered is used later by the scripts that perform the installation of the database segment.

The `PostInstall` descriptor file is used to set up the installation environment, start the RDBMS if necessary, and invoke the scripts that perform the installation of the database segment.

For database segments, the `ReleaseNotes` descriptor should show how applications, operating system groups, and database roles are associated. Developers should also provide the database schema, including

its dependencies. In addition to any narrative information in this file, developers should include comments on their schema, data objects, and data elements as part of their database build.

The `Requires` descriptor must identify the required RDBMS and version. It must also identify all dependencies on other database segments.

As with data segments, database segments have a scope associated with them. The scope is specified in the `Database` segment descriptor, as explained in subsection 5.5.2.9.

`install` Subdirectory

The `install` subdirectory contains the scripts to install and then create the database segment. It includes all of the DDL scripts that create the database objects for the segment. There are two sets of DDL scripts in this directory. The first set allocates storage for the database, creates the database owner, and defines the roles associated with the database segment. It must be executed by a DBA. The second set creates all database objects (tables, views, indexes, sequences, constraints, triggers, etc.) that make up the database. This set must be executed by the database owner.

The naming conventions to be used for database definition scripts and the structure of those scripts are discussed in Chapter 4.

`data` Subdirectory

The `data` subdirectory contains any data files used to load the database. Data fill may also be provided in a separate data segment if developers wish or need to keep the fill separate.

Several methods for loading data, depending on data size, are discussed in subsection 5.9.3.

`bin` Subdirectory

The `bin` subdirectory contains any scripts or other executables used to load data from the data files into the database. It may also contain any applications that support unique database administration requirements for that database segment.

`DBS_files` Subdirectory

The `DBS_files` subdirectory contains the DBMS-controlled data files that make up the storage for the database. This directory is owned by the DBMS, not the segment. The data files are created during the installation of the segment, normally in the `PostInstall` process. Directory ownership must be transferred to the DBMS before the data files are created. Note that this does not allow developers to stipulate disk architecture.

5.4.6 Patch Segment Types

The COE supports the ability to install field patches on an installed software base. A patch segment permits the replacement of one or more individual files, including those of the operating system. It does *not* refer to overwriting a portion of a file, as is sometimes done to patch a section of binary code.

Patches are created in a segment whose directory name is the directory name of the affected segment followed by a ".", followed by the letter "P", followed by the patch number. Figure 5-11 shows an example patch segment directory structure for applying patch 5 to an ASW segment. The subdirectory `SegDescrip` is required, but the remaining subdirectories are patch-dependent. The example illustrates a situation in which scripts, executables, and data files are to be updated by installation of a single patch segment.

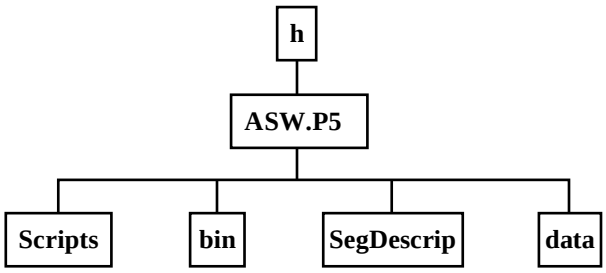


Figure 5-11: Example Patch Directory Structure

The installation software loads patches underneath the affected segment in a subdirectory called `Patches`. Figure 5-12 shows the result of loading patch 5 from Figure 5-11. This approach makes it easy to find and identify what patches have been applied to a segment. It also makes it easy for the installation software to automatically remove patches when a segment is replaced by a later update. If there is insufficient room to physically load the patch underneath the `Patches` subdirectory, the patch is loaded on the first available disk partition. A symbolic link is then created to preserve the logical structure shown. Also note that when installed, the resulting subdirectory name of the patch for this example is `P5`, not `ASW.P5`.

As patches are installed and removed, the descriptor file `Installed` in the segment descriptor directory for the affected segment is updated to reflect what patches are installed and removed, the date and time, the installer's name, and the platform from which the work was done.

When a patch is installed, it is the patch segment's responsibility to perform whatever operations are necessary to replace files. In the example shown, the `PostInstall` script must copy files from `Scripts`, `bin`, and `data` as required to update files in the existing ASW segment.

To facilitate patch removal, the `PostInstall` program may create compressed copies of files before they are modified and put them underneath the patch subdirectory (e.g., the `ASW/Patches/P5` subdirectory in this example). In this way, a `DEINSTALL` descriptor simply needs to copy the files from the patch subdirectory to their original place and decompress them to restore the system to the pre-patch state. If the files being replaced are large, this may require too much disk space to store the original files. In such cases, the patch segment should be designated as a permanent patch and not make copies. A patch segment is considered to be permanent if the patch segment does not include a `DEINSTALL` descriptor.

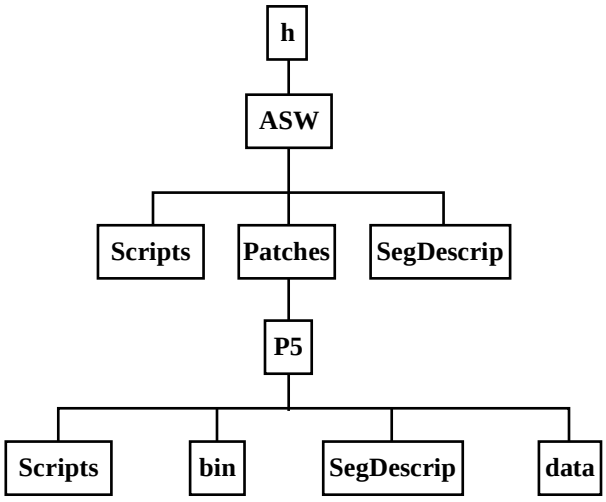


Figure 5-12: Example Installed Patch

The COE installation software assumes that higher numbered patches must be removed before a lower numbered patch can be removed. For example, patch 2 cannot be removed until patch 5 is removed. However, if patch 5 cannot be removed - because there is no `DEINSTALL` descriptor for patch 5 - patches 1 and 2 cannot be removed either. The only way to remove them is to remove the entire segment.

DII compliance requires that:

- Patch segments shall be named *`SegDir`*`.P`*`number`* where *SegDir* is the assigned directory name for the segment to be patched, and *number* is a sequential patch number.
- Patch segments shall perform the necessary operations to replace files through the `PostInstall` script.
- Permanent patch segments shall be designated by the absence of a `DEINSTALL` script.

Patch segments can also be used to make updates to a database segment prior to the release of a new database segment that incorporates the patch. The patch segment structure will be the same as the database segment being patched, and the patch name follows the same conventions as for any other patch segment.

Any objects, scripts, etc. that are being updated will be in the same location under the patch segment directory as the corresponding original is under the database segment directory. `PostInstall` will be used to backup the original and copy the new file to the database segment directory. The patch segment will have the same owner as the database segment being patched.

Any changes to executables provided with the patch will be implemented in the same manner as patches to other software segments. Any changes to the database provided with the segment will require an analysis to determine application segment dependencies. Changes to the database must be coordinated with application segment developers.

If the patch segment is making any changes to the database objects, its developers are responsible for preserving the information those objects currently contain, together with restoring any permissions that have been granted on the objects. This usually requires extracting and saving the records from the objects being modified, making the schema changes, and then reloading their data. That portion of the patch segment must be implemented in a manner that allows it to be restarted or re-executed without data loss in the event of system or media failure during the patch installation.

5.4.7 Aggregate, Parent, and Child Attributes

It is sometimes convenient for a collection of segments to be treated as an indivisible unit. The aggregate attribute provides this capability and the collection of segments are called an aggregate segment. One, and only one, segment is designated as the *parent* segment and the remaining segments are designated as *children*. Parent and child segments are designated as members of an aggregate in the `SegName` descriptor file. The child segment must list its parent segment in `SegName` (but not in `Requires`), while the parent segment must list each child (in `SegName` but not `Requires`) in the aggregate. See subsection 5.5 for the segment descriptor information required to do this. Each segment within the aggregate is packaged according to its segment type as described in preceding subsections.

The parent segment plays a special role in the aggregate. During installation with the segment installer, only the parent segment is "seen" by the operator. Child components are not displayed as selectable items, but are automatically loaded with the parent. Therefore, the segment name and release notes associated with the parent segment should be carefully chosen to be properly descriptive of the aggregate.

The parent segment is the first segment loaded from the aggregate. Child segments are loaded next in the order listed by the parent segment. Because of this, child segments may specify a dependency on the parent, but shall not specify dependencies upon one another.

In some situations, a child segment in an aggregate should be loaded conditionally. That is, the child should only be loaded if it is not already on disk, or only if it is a later version. An example of this situation is if a collection of segments created by a single developer must use the same executable. One approach would be to create the common executable and put it into its own separate segment. Then all the remaining segments would need to state a dependency upon it. An alternative approach, supported here, is to package the common executable as a child segment that is to be conditionally loaded and placed in an aggregate with each segment that needs it. The conditional load capability is specified by the `$LOADCOND` keyword in the child segment's `SegName` descriptor (see subsection 5.5.1.10).

The COE requires that each segment include a `Security` segment descriptor. This file is used primarily as a documentation aid and is used by the installer tool to indicate which segments are classified at what level. The security level of the parent segment must dominate that of the child segments. For example, if a child segment has a SECRET classification, then the parent segment must have a SECRET or higher classification. The segment developer must ensure that each segment in the aggregate is compatible for the hardware platform. `VerifySeg` will check for this condition and reject an aggregate with incompatible hardware platforms specified.

Disk space required is specified by each individual segment, not by the aggregate parent. The COE installation tools may load parent and child segments on different disk partitions, depending upon space available at install time. During installation, the space reported to the installer takes into account whether or not the aggregate includes a conditional load child, and whether or not the segment is already on disk. That is, the installer tool reports the *additional* space required on the disk to load the selected segment(s).

DII compliance requires:

- One and only segment in the aggregate shall be designated as the parent segment.
- Child segments may specify a dependency on the parent, but shall not specify dependencies upon one another.
- The security level of the parent segment shall dominate the security level of all child segments.
- Segments within an aggregate shall be consistent with regard to the hardware platform specified.
- Segments shall individually specify their own disk space requirements.

5.4.8 COE-Component Attribute

Segments authorized by the DII COE Chief Engineer may specify the attribute of being a COE-component segment. COE-component segments are similar to aggregate segments in that one segment serves the role of a parent segment and all others are children to that parent. The parent segment is similar to an account group segment which is affected by a collection of child component segments. However, there are important differences between COE-component segments and aggregate segments, and between the parent COE-component segment and account groups.

- At installation time, a segment identified as a COE component must have an authorization key[42] (see the `$KEY` keyword) specified or else the segment will be rejected.
- Exactly one segment is designated as the parent COE component for the entire system. This is the segment whose directory is `/h/COE`.
- Child COE-component segments are not loaded unless they are required. That is, a child COE-component segment will not be loaded unless there is another segment which expresses a dependency upon it.
- COE-component segments are organized into a very specific structure.
- The parent COE-component segment does not set up a runtime environment. It sets up a baseline environment which is inherited by all account groups.

Figure 5-13 shows the directory structure for COE-component segments. Since COE components form the foundation for the entire system, they are collected together in a single place and are validated more rigorously during segment development, integration, and installation. Special processing, as explained below, is performed on the COE components because of their unique position within the architecture.

The `SegDescrip` subdirectory, required for all segments, underneath `/h/COE` refers to the collection of COE components as a whole. Segments designated as child COE components are loaded in the subdirectory `/h/COE/Comp`. Each child COE-component segment has its own `SegDescrip`, `bin`, `Scripts`, and `data` subdirectory as appropriate. If insufficient space exists to load the COE component directly under `/h/COE/Comp`, a symbolic link is created to where the segment was actually loaded.

Environment files underneath `/h/COE/Scripts` are included by every account group so that they are automatically inherited by every segment. The file `.cshrc.COE` sets the path environment variable so that `/h/COE/bin` is first in the search path before any other segments. Environment extensions for child COE components are handled differently than environment extensions for other segments. As child COE-component segments are installed, environment extension files located underneath the child COE component's `Scripts` subdirectory are textually inserted directly into the appropriate file underneath `/h/COE/Scripts`. This insertion is performed automatically by the installation tools. This is done to avoid the runtime overhead of executing several `source` statements to pick up child segment extensions.

Child COE-component segments shall not alter the `path` environment variable. It is not necessary to do so because as child COE components are loaded, the installation tools create a symbolic link underneath `/h/COE/bin` to where the executables were actually loaded. This is done so that the search path contains

[42] To preserve backwards compatibility, segments which are already authorized as COE-component segments are not required to use the `$KEY` keyword for this *I&RTS* release. However, they are required to migrate to this approach. In the interim, a legacy segment identified as a COE-component segment which does not use the `$KEY` keyword is compared against a table containing the names of authorized COE-component segments. If it does not match, the segment is rejected. All new COE-component segments must use the `$KEY` keyword.

only one entry for the COE, regardless of the number of actual segments comprising the installed COE. This approach mandates that all COE-component segments use the segment prefix to name executables. `VerifySeg` will issue a warning for COE-component segments that do not meet this requirement, but in a future release it will strictly fail such a component.

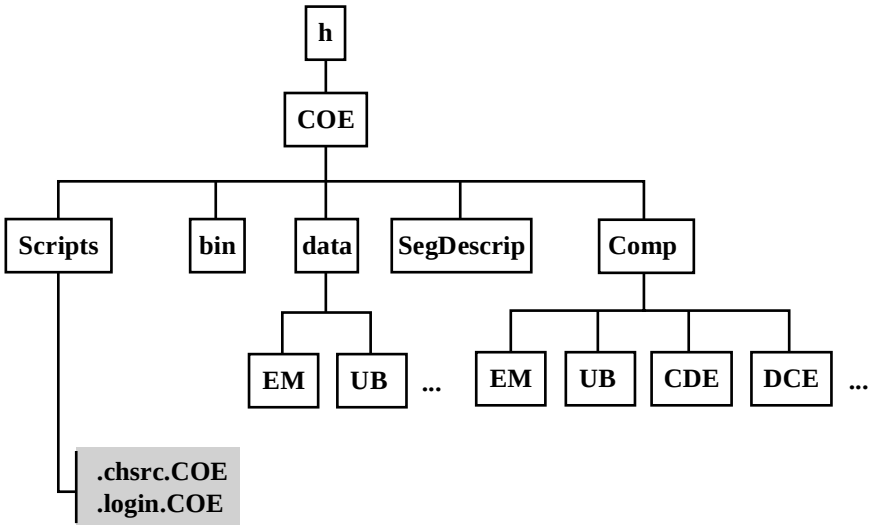


Figure 5-13: COE-Component Segments Directory Structure

Symbolic links are also created underneath `/h/COE/data` to point to the child COE component's `data` subdirectory. The installation tools automatically delete these symbolic links when a COE-component segment is deinstalled.

To summarize DII compliance requirements:

- COE components shall be authorized by the DII COE Chief Engineer. They will be issued an authorization key that the developer shall specify in the segment with the `$KEY` keyword.
- Child COE components shall not alter the `path` environment variable.
- COE components shall use the segment prefix to name all executables.
- Child COE components shall use the segment prefix to name all public symbols contained in files within the segment's `Scripts` subdirectory.

5.4.9 DCE Attribute

The DII COE supports both DCE server and DCE client applications. Servers are designated with the `DCEServerDef` segment descriptor (see subsection 5.5.2.11) while clients are designated with the `DCEClientDef` segment descriptor (see subsection 5.5.2.10). Segments, whether a DCE server or a DCE client, must indicate the DCE attribute or else the `VerifySeg` tool will generate a fatal error when processing DCE-related segment descriptors.

5.4.10 Web Attribute

Segment types that have the Web attribute are either Web servers or Web-application segments (e.g., Web clients). By definition, Web servers are also COE-component segments, so they have that implied attribute as well. Web applications may or may not be COE components, and so must indicate explicitly whether or not they are. This is described in subsection 5.5.1.10.

Web applications can only be installed on a platform that already has a Web server loaded on it. Therefore Web applications must be designed so that they can access other COE services that may be located on another platform, possibly even behind a firewall. This allows sites to isolate the main COE-based system from the Web server by firewalls or other security-related techniques.

Other than specifying the Web attribute, no additional segment descriptors are presently required beyond those identified for all other segments.

5.4.11 Generic Attribute

The Generic attribute is provided to allow a segment to indicate that it should be automatically made a member of all "regular" account groups. This means that the segment, unless it indicates otherwise, will be made a participant of all account groups except those which are character-interface-based (e.g., `CharIF`) or accessed through remote execution account groups such as `RemoteX`.

This capability is provided for two reasons. First, some segments should be made a member of virtually every account group. An example is a Web browser which is set up to provide access to HTML help pages. Such a segment should be a member of the following:

- the System Admin account group
- the Security Admin account group
- the Database Admin account group
- the operator account group (e.g., GCCS, ECPN).

It is convenient that this happen automatically without the need for the segment to explicitly list every account group it is to be a member of. Such segments do not need to express any affected account group in the `SegName` descriptor.

Second, some segments developed for one system may be generally applicable to other mission systems, yet this may not have been realized when the segment was created. Using the Web browser example, if it is packaged for GCCS and it states `GCCS` is the affected account group, the segment's `SegName` descriptor will need to be modified to use it for a different system such as ECPN or GCSS. Declaring the segment to have the generic attribute avoids this problem.

There are some special points to note about segments which declare the generic attribute:

- The segment is automatically added to *every* account group except `CharIF` and `RemoteX`.

- Site administrators can establish user profiles to deny an operator access to the generic segment, even if it is a member of an account group.

- The generic segment is only stored on the disk once, regardless of how many account groups it is made a member of.

- Generic segments may exclude account groups by listing the groups to exclude with the `$EXCLUDE` keyword in the `SegName` descriptor.

- The generic attribute may be combined with other segment attributes. Subsection 5.5.1.10 states which attributes may be combined.

5.4.12 Segment Dependencies

Segments specify dependencies upon one another through the `Requires` descriptor, and, for database segments with database dependencies, the `Database` descriptor. However, the COE does not allow circular dependencies. That is, a situation where Seg A depends upon Seg B, Seg B depends upon Seg C, and Seg C depends upon Seg A is strictly forbidden.

Components of an aggregate may have dependencies upon other components within the same aggregate and such dependencies could lead to the circular situation just described. But since components of an aggregate are always loaded together as a unit, this does not pose a problem. Child components of an aggregate must *not* specify dependencies upon one another in the `Requires` file, even if they do indeed have such dependencies. Likewise, the parent segment must *not* specify a dependency on children within the aggregate. An aggregate of database segments cannot have circular database dependencies among the segments or there will be no valid database creation order.

5.5 Segment Descriptors and Descriptor Files

This section details the contents of the segment descriptor files. These files are the key to providing seamless and coordinated systems integration across all segments. Adherence to the format described here is required for all segments to ensure DII compliance. This enables automatic verification and installation of segments.

The software tool `VerifySeg` must be run during the development phase to ensure that segments properly use segment descriptor files. The software tool `MakeInstall` uses information in segment descriptor files to compress and package segments in a format suitable for installation from tape, from a disk-based LAN segment server, from a remote site, or from other media. At installation time, the installation tools use segment descriptor information to make the COE changes required (e.g., update menu files) so that software components are available to the user.

Some segment information is contained within individual files while other segment information is collected into a single file, `SegInfo`. Segment descriptors which are contained in their own separate file are discussed in subsection 5.5.1 while segment descriptors that are contained within the `SegInfo` file are discussed in subsection 5.5.2. `SegInfo` is an American Standard Code for Information Interchange (ASCII) file (similar to a Windows `.INI` file) with multiple sections containing segment descriptor information.

Table 5-4 lists each of the descriptor files and which are required, optional, or not applicable for each segment type. Table 5-5 lists the same information for segment descriptor sections within the `SegInfo` descriptor file. The `VerifySeg` tool will display these two tables when the `-t` flag is given on the command-line so that the latest information from these two tables is available online.

A `SegInfo` segment section begins with a single line of the form

```
[section name]
```

where *section name* is chosen from the list in Table 5-5. A section continues until another section name is encountered, or the end of the file is reached. A section may appear only once within the `SegInfo` file, but the order in which sections appear is unimportant. Section names are not case sensitive.

If a section name that the tools do not recognize is encountered, a check is made to see if a helper function is available to process the section. If so, the helper function is invoked, otherwise an error is issued. Appendix C describes which tools accept helper functions. Creation of a helper functions require authorization by the DII COE Chief Engineer.

Descriptor File	COTS	Acct Grp	S/W	Data	DB	Patch
DEINSTALL	O	O	O	O	O	O
FileAttribs	O	O	O	O	O	O
Installed	I	I	I	I	I	I
PostInstall	O	O	O	O	O	R
PreInstall	O	O	O	O	O	O
PreMakeInst	O	O	O	O	O	O
ReleaseNotes	R	R	R	R	R	R
SegChecksum	I	I	I	I	I	I
SegInfo	R	R	R	R	R	R
SegName	R	R	R	R	R	R
Validated	I	I	I	I	I	I
VERSION	R	R	R	R	R	R

R - Required O - Optional N - Not Applicable
I - Created by Integrator or Installation Software

Table 5-4: Segment Descriptor Files

Section	COTS	Acct Grp	S/W	Data	DB	Patch
AcctGroup	N	R	N	N	N	N
*AppPaths	N	O	O	N	N	N
COEServices	O	O	O	O	O	O
Community	O	O	O	O	O	O
Comm.deinstall	O	O	O	O	O	O
Compat	O	O	O	O	O	N
Conflicts	O	O	O	O	O	O
Data	N	N	N	R	N	N
Database	N	N	O	N	R	O
DCEClientDef	O	N	O	N	N	N
+DCEServerDef	O	N	O	N	N	N
Direct	O	O	O	O	O	O
FilesList	R	O	O	O	O	O
Hardware	R	R	R	R	R	R
Help	O	O	O	O	O	O
Icons	O	R	O	N	N	O
Menus	O	R	O	N	N	O
**Network	O	N	O	O	N	O
Permissions	N	O	O	N	N	O
Processes	O	O	O	N	N	O
*Registry	O	O	O	O	O	O
+ReqrdScripts	N	R	O	N	N	N
Requires	O	O	O	O	O	O
Security	R	R	R	R	R	R
SharedFile	O	O	O	N	N	O

R - Required O - Optional N - Not Applicable
*** - NT platforms only + - UNIX platforms only**
**** - COE Component Segments Only**

Table 5-5: SegInfo Segment Descriptor Sections

Certain general characteristics are common to all files or sections listed in these two tables:

1. All descriptor files are ASCII data files, except for those which are executables (e.g., `PostInstall`, `PreInstall`, `PreMakeInst`, and `DEINSTALL`) which may be script files or compiled code. Regardless of platform, the descriptor files may have an optional file extension. The `.TXT` file extension is permitted for each descriptor file except `DEINSTALL`, `PostInstall`, `PreInstall`, and `PreMakeInst`. These are actually executables and may have a `.BAT` extension (for batch files), a `.EXE` extension (for compiled code), or no extension at all. The file extensions are optional, but developers should conform to standards on the platform for which the segment is targeted.

2. In describing syntax, options which may appear exactly once are delimited by brackets (i.e., "[ ]"), while options that may appear multiple times are delimited by braces (i.e., "{ }"). The "|" (boolean exclusive or) symbol is used to indicate a selection of one item from a list of choices. The delimiters are not entered into the actual descriptor file.

3. Descriptor files may contain comments. Comments are delimited by using either the standard C convention[43] (e.g., delimited by /* */), or on a line by line basis using the # character. *C style comments may not be nested.* C style comments may not be used in `PostInstall`, `PreInstall`, `PreMakeInst`, or `DEINSTALL` since these are executable scripts. (These may also be compiled programs instead of scripts, although scripts are recommended because they can be examined at integration time for potential problems.)

4. Blank lines may be used freely and are ignored unless they are within a block of text for insertion, replacement, etc. Blank lines are ignored when searching for a block to delete or replace. Similarly, blanks, tabs, and other whitespace are ignored unless they are part of a block to insert or replace.

5. When a block of text is required, such as in adding a block of text to a community file, the characters "{" and "}" are used as block delimiters.

6. Keywords inside a descriptor file are always prefixed with the "$" character.

7. C style `#ifdef`, `#else`, `#elif`, `#endif`, and `#ifndef` constructs may be used in descriptor files, along with the standard C boolean operators. These constructs may not span segment descriptor sections. The constants which may be used in these constructs are defined in subsection 5.3.

8. During installation, the COE installation software sets up to five environment variables: `INSTALL_DIR` is the absolute pathname to where the segment will be loaded (`PreInstall`) or was loaded (`PostInstall`). `MACHINE_CPU` and `MACHINE_OS` are set to describe the type of platform on which the software has been loaded. Valid values for these environment variables are listed in subsection 5.3. `SYSTEM_ROOT` (for NT only) is set to point to the directory where Windows is installed. `COE_TMPSPACE` is the location of temporary space allocated for the duration of segment installation.

9. Parameters which follow a keyword are given on the same line as the keyword and are separated by colons. The exception to this rule is when the keyword signals the beginning of a variable length list. For example,

```
$PATH:/etc
```

[43] This should not be misunderstood as stating a preference for C/C++ over Ada or any other language. The comments referred to are placed in data files, not executable code. C style comments were selected because they allow a block of text to be commented out by surrounding the block with a single "/* */" pair instead of including a comment token on each line.

specifies a pathname while

```
$LIST
f1
f2
f3
```

specifies a list of files.

10. Some segment descriptors, such as the `Requires` descriptor, specify the name of another segment that the COE installation tools must search for at install time. To speed up the search process, segment names are expressed in the form

    ```
    segment name:prefix:home dir:[version:{patches}]
    ```

where *segment name* is the name of the segment, *prefix* is the segment's prefix, *home dir* is the segment's expected home directory, while *version* and *patches* are optional. *home dir* is searched first, and if the segment name found there is the same as that specified, a match is declared successful. If *home dir* does not exist, is not a segment, or the segment name does not match, an exhaustive search is performed on all segments on all mounted disk partitions.

11. (NT) When a disk drive needs to be specified in a filename, the filename must be enclosed in double quotes. This is required so that the tools can distinguish between use of ':' as a field delimiter for descriptors, or as a separator between a disk drive name and a pathname.

12. Some segment descriptors allow a version number or patch level to be specified. See the previous `Requires` example. If no version number is specified, any version found is successful. If a version number is specified, an ordinary lexical comparison of primary version numbers is made with zeroes inserted for any missing digits. For example, a version number such as 3.4/SunOS-4.1.3 is truncated to just the primary version number which is then expanded to be 3.4.0.0 for comparison purposes.

13. Some descriptor file features require prior Chief Engineer approval, or are restricted to COE-component segments. These are described in the sections which follow and generally require the `$KEY` keyword to be specified in the applicable section. This keyword requires an authorization key provided by the Chief Engineer. The authorization key is based on several segment attributes including segment name, segment prefix, and the section name to which it applies. The format of the `$KEY` keyword is

    ```
    $KEY:permit requested:authorization key
    ```

where *permit requested* is the keyword or section name the key applies to, and *authorization key* is the key given to the developer by the Chief Engineer. A separate authorization key is required for **each** permit requested.

14. Certain keywords or section names may be applicable to one platform but not another. These are noted in the discussion below. If the tools encounter a keyword that is not appropriate for a platform, a warning will be generated and the keyword or section will be ignored.

15. A segment is considered to be a permanent segment if the `DEINSTALL` descriptor is not provided. This means that the installation tools will prevent a permanent segment from being deleted, but it may be upgraded by loading a newer version of the segment.

DII compliance requires the following:

- Segments shall include all required files shown in Table 5-4. (`VerifySeg` will fail a segment that does not include a required descriptor file or descriptor section.)

- Segments shall fully specify all dependencies and conflicts through the `Requires` and `Conflicts` descriptors. (Circular dependencies are not allowed.)

- Segments shall fully specify disk and memory requirements (memory may be omitted for data segments) in the `Hardware` file.

- Segments shall not use `PostInstall`, `PreInstall`, `PreMakeInst`, or `DEINSTALL` to make modifications that the COE installation software will make. Of particular importance is that segments shall not delete the segment directory during a `DEINSTALL` script.

- Segments shall use the `ReleaseNotes` file to convey information meaningful to an operator, not the system integrator. `ReleaseNotes` files shall not include company names, names of individuals, nor software trouble report numbers.

- Segments shall specify a version number and date in the `VERSION` descriptor file and shall increment the version number for each subsequent release. Version numbers shall fully comply with the requirements stipulated in Chapter 3 of this document.

5.5.1 Segment Descriptor Files

This subsection describes all the segment descriptors that are contained in individual files.

5.5.1.1 DEINSTALL

The `DEINSTALL` descriptor file is an executable, either a script or a compiled program, that is invoked by the installation software when the operator has elected to remove a segment. This may occur by explicitly selecting a segment to remove or by electing to install a new version of the segment. `DEINSTALL` should perform actions such as shutting down segment-owned background processes prior to segment removal. Operations performed in preparation for a segment update should normally be done in `PreInstall`, while `DEINSTALL` is used when the segment is to be "permanently" removed from the system.

If this file does not exist, the segment is assumed to be permanent and cannot be removed except when installing a new version. If a new version is installed and this file does not exist, the installation software will use the information in the descriptor directory to undo changes made by the previous installation of the segment and then simply delete the directory.

For security reasons, the `DEINSTALL` script is not run with root-level privileges, unless the `$ROOT` keyword is given in the `Direct` descriptor. Note that the `$KEY` keyword must also be specified in the `Direct` descriptor to acquire root-level privileges.

5.5.1.2 FileAttribs

The `FileAttribs` descriptor file allows a segment to specify the attributes (owner, read/write permissions, group) for each file in the segment. It is created by the tool `MakeAttribs` (see Appendix C). The installation tools, just prior to `PostInstall`, will use information in this file to set file attributes.

`FileAttribs` has certain restrictions due to security and segment integrity considerations. The following will be ignored:

- Files within the `SegDescrip` subdirectory
- Files outside the segment

- Requests to set root ownership
- Requests to set UNIX "sticky bits" (e.g., `chmod 4644`)

If `FileAttribs` is not provided by the segment, the installation tools will automatically do the following for all except COTS segment types:

- `chmod 554` for all files in the `bin` subdirectory
- `chmod 664` for all files in the `data` subdirectory
- for account groups, set owner to the same group id as specified in the `AcctGrps` descriptor for all subdirectories except `SegDescrip`
- for other segment types, set owner to the same group id as the affected segment for all subdirectories except `SegDescrip`.

5.5.1.3 Installed

The installation software creates the file `Installed` as segments are loaded. The file specifies the segment that was loaded, the date and time of the installation, which platform was used to do the installation, and the version number of the software used to do the installation. This file is located underneath the segment descriptor directory.

5.5.1.4 PostInstall

Most of the work required to install segments is performed by the COE installation software through information contained in the descriptor directory. However, additional segment-dependent steps must sometimes be performed. `PostInstall` is an executable, either a script or a compiled program, that segment developers may provide to handle segment-specific installation functions *after* the segment has been copied to disk and installed by the COE. During installation, `PostInstall` may invoke functions (e.g., prompt the user) described in Appendix C.

The `PostInstall` descriptor must *not* do any operations that are performed by the COE installation software. For security reasons, the `PostInstall` script is not run with root-level privileges unless the `$ROOT` keyword is given in the `Direct` descriptor. Note that the `$KEY` keyword must also be specified in the `Direct` descriptor before root-level privileges will be granted.

5.5.1.5 PreInstall

The `PreInstall` descriptor file is identical to `PostInstall` except that it is invoked by the installation software *before* the segment is loaded onto the disk. It must *not* do any operations that are performed by the COE installation software. For security reasons, the `PreInstall` script is not run with root-level privileges, unless the `$ROOT` keyword is given in the `Direct` descriptor. Note that the `$KEY` keyword must also be specified in the `Direct` descriptor before root-level privileges will be granted.

5.5.1.6 PreMakeInst

`PreMakeInst` is an optional executable program or script that is invoked by the `MakeInstall` tool. Its purpose is to allow a segment to perform "cleanup" operations, before `MakeInstall` writes the segment to the distribution media. Example cleanup operations include:

- deleting temporary files
- ensuring no "core" or other "garbage" files are in the segment
- ensuring no compiler "scratch" files, such as temporary intermediate object files, are in the segment.

`MakeInstall` sets the environment variables `INSTALL_DIR`, `MACHINE_CPU`, and `MACHINE_OS` prior to invoking `PreMakeInst`.

5.5.1.7 ReleaseNotes

Use the ASCII file `ReleaseNotes` to provide information useful to an operator in understanding the new functionality being provided by the segment or the problems being fixed, and a system administrator responsible for installing segments. It is *not* a help file, nor is it information targeted to the system integrator. Therefore, it must not refer to problem report numbers, version[44] numbers, release dates, individuals or companies, point of contact, or similar information. (This type of information is contained elsewhere, such as in the `VERSION` file, and duplication of information may lead to conflicting or confusing information for the operator.) The `ReleaseNotes` file must not contain any tabs or embedded control characters.

An example of a "poor" `ReleaseNotes` file is

```
Release: 5.6.3
Point of Contact: John Doe, Tritron Company
Phone: (619) 555-1234

1. Implemented NCR #302
2. Added check for memory overflow
3. Fixed problem with double scrolling in STR #307
```

An example of a "good" `ReleaseNotes` file is

```
This release fixes two known problems:

(a) Calculation of range and bearing for polar latitudes
has been corrected

(b) Display of garbled latitude/longitude in the Track Summary
display for ownship has been corrected

The following new features are added with this release

1. Search and Rescue TDA added.
2. Option added to restrict operator deletion of comms msgs.
```

The `ReleaseNotes` is also a good place to convey information to the sites about any COTS features that are disabled or that may have restrictions on releasability to foreign nationals.

5.5.1.8 SegChecksum

The file `SegChecksum` is an optional file created by integration software. It contains information necessary for the System Administrator software to perform an integrity check on the installed software. If the file does not exist, the integrity check cannot be performed on the segment.

[44] The `COEInstaller` contains a "print" button which allows the release notes to be printed out. It automatically appends the segment name, and version and date (from the `VERSION` descriptor) to the output. This tool also has a button which allows a user to view the release notes on the screen, including release notes for child segments in an aggregate.

5.5.1.9 SegInfo

SegInfo is an ASCII descriptor file which contains segment information in one or more segment descriptor sections. Table 5-5 lists the possible sections.

5.5.1.10 SegName

The SegName descriptor file provides the following information:

- segment type ($TYPE keyword)
- segment name ($NAME keyword)
- segment prefix ($PREFIX keyword)
- segment attributes ($TYPE keyword)
- optional aliases for this segment ($EQUIV keyword)
- conditional loading requirements ($LOADCOND)
- company and product name (For UNIX, this is for documentation only. For NT, these are added to the registry.)
- if applicable, affected account group, or affected segment for patches ($SEGMENT keyword)
- if applicable, name of parent or child segments ($PARENT, $CHILD keywords)

The keywords $TYPE, $NAME, and $PREFIX are required for each SegName descriptor file. Additional keywords required depend upon segment type. COE-component segments may not contain $SEGMENT, $PARENT, or $CHILD keywords. All other segments must have one $PARENT line, one or more $CHILD lines, or one or more $SEGMENT lines.

$COMPANY_NAME:*string1*
$PRODUCT_NAME:*string2*

These two keywords are intended for use with COTS products on NT platforms. If either keyword is used, both are required. They cause the installer to insert the company name (*string1*) and product name (*string2*) in the registry entry

```
SOFTWARE\company name\product name
```

Note: These keywords may be present for a UNIX platform, but are presently ignored. They are intended for future use in UNIX.

$EQUIV:*name*:*prefix*

This keyword, which may appear multiple times, allows a segment to define aliases. It is intended to help legacy segments migrate from an earlier COE (e.g., JMCIS or GCCS COE) to the DII COE. It is primarily intended for account group segments, but may be used for other segments as well. *name* is the desired alias and *prefix* is the alias segment prefix.

This keyword allows a segment from a legacy system to be loaded under an equivalent account group without the need to modify the legacy segment's dependency statements. For example, assume that SegA was originally developed for JMCIS and that it states in its segment descriptors a dependency on an account group whose name is JMCIS. Assume that the legacy segment prefix was JMC. Assume that SegB was developed for the GCCS account group. Finally, assume that SegA and SegB are to be loaded on a new system under an account group whose name is New Acct Group and whose segment prefix is NAG. Then the keyword entries

```
$NAME:New Acct Group
```

```
$PREFIX:NAG
$EQUIV:JMCIS:JMC
$EQUIV:GCCS:GCCS
```

allow SegA and SegB to be loaded properly even though they state a dependency on segments, JMCIS and GCCS, that do not exist in the new system.

$EXCLUDE:*name*:*prefix*:*home dir*

This keyword is used to indicate an account group that a generic segment is to be excluded from. *name* is the name of the account group, *prefix* is the account group's segment prefix, and *home dir* is the assumed location of the account group's assigned directory. This keyword can only be used with segments that specify the GENERIC attribute. The CharIF and RemoteX account groups are automatically excluded.

$KEY:COE:*key*

This keyword is required for all segments that have the attribute COE CHILD, COE PARENT, or WEB SERVER. *key* is the authorization key obtained from the DII COE Chief Engineer. For backwards compatibility, existing COE-component segments are "grandfathered" and may omit this keyword for now. However, existing segments should be modified to use this keyword to ensure future compatibility.

$LOADCOND

This keyword, which accepts no parameters, is used to indicate that a child segment in an aggregate is to be conditionally loaded. The child segment is loaded only if the segment does not already exist on the disk or if the child segment is a later version than one already on the disk. If this keyword is used, the segment must also have the CHILD or COE CHILD attribute or else an error is given. This capability is not required for any other type of segment because the installer tool already checks to be sure an earlier version is not unintentionally being loaded over a later version.

$TYPE:*segment type*[:*attribute1*:*attribute2*:...]

where valid *segment types* are

```
COTS
ACCOUNT GROUP
SOFTWARE
DATA
DATABASE
PATCH
```

and valid segment *attributes* are

```
AGGREGATE
CHILD
COE CHILD
COE PARENT
DCE
WEB SERVER
WEB APP
GENERIC
```

AGGREGATE is used to indicate that the segment being defined is the aggregate parent segment. It is valid only for account group, data, and software segment types. Aggregates must list one or more child segments with the $CHILD keyword. The COE does not allow an aggregate of aggregates. That is, it is not valid for Aggregate A to have a child B which is also an aggregate.

CHILD is used to indicate that the segment being defined is an aggregate subordinate segment. The parent segment must be listed using the $PARENT keyword.

COE PARENT is used to indicate that the segment being defined is the primary COE segment. Its home directory will be /h/COE.

COE CHILD is used to indicate that the segment being defined is a COE-component segment other than the parent. The installation tools will verify that the segment is an authorized COE component and if not will reject the segment. This is done through the $KEY keyword.

DCE is used to indicate that this segment is a DCE server or a DCE client application. This attribute *must* be specified to use any DCE-related segment descriptors.

WEB SERVER is used to indicate that this segment is a Web server and a COE-component segment.

WEB APP is used to indicate that this segment is a Web-based application segment.

GENERIC is used to indicate that this is a generic segment that should be added to the account groups as described in subsection 5.4.11.

Segment types are mutually exclusive; only one segment type may be given. Segment attributes are also mutually exclusive, except for DCE, Web and GENERIC attributes as follows:

- DCE may be combined with AGGREGATE, CHILD, or COE CHILD.
- WEB SERVER may be combined with AGGREGATE, CHILD, or COE CHILD.
- WEB APP may be combined with AGGREGATE, CHILD, or COE CHILD.
- GENERIC may be combined with all other attributes except WEB SERVER and COE PARENT.

For example, a generic Web mission application that is a child component of an aggregate would be expressed as

```
$TYPE:SOFTWARE:CHILD:WEB APP:GENERIC
```

The order in which attributes are listed is unimportant.

> **Note:** There are two important considerations with respect to aggregate segments. First, when a change is made to any segment within an aggregate, the version number of the parent must be updated to reflect that a change has occurred. If a child segment was modified, then the version number of the child must be updated as well. This is in keeping with good configuration management practices. Secondly, the parent segment in the aggregate must specify the version number for each child in the aggregate. See the $CHILD keyword. This is required to ensure that the child components are the exact version that the parent is expecting.

$NAME:*name*

where *name* is a string of up to 32 alphanumeric characters. Embedded spaces may be used for readability, but the string must not contain tabs or other control characters.

$PREFIX:*prefix*

This keyword establishes the segment's assigned prefix, *prefix*.

$SEGMENT, $CHILD, $PARENT

The syntax for $SEGMENT and $PARENT is the same:

```
keyword:name:prefix:home dir
```

The syntax for $CHILD is

```
$CHILD:name:prefix:home dir:version
```

where *version* must[45] include all 4 digits of the version number and must match the version number in the VERSION descriptor for the child segment that is referenced.

This descriptor file may contain one and only one $PARENT keyword. Multiple affected segments or child segments may be listed by listing each segment on a separate line.

> **Note:** Do not confuse the attribute CHILD with the $CHILD keyword. The $CHILD keyword is used to indicate a list of subordinate segments in the parent of an aggregate segment. The CHILD attribute is used to indicate that a segment is the subordinate segment in an aggregate whose parent is identified with the $PARENT keyword.

5.5.1.11 Validated

The COE requires strict adherence to integration and test procedures to ensure that a fielded system will operate correctly. To facilitate integration and testing, the VerifySeg tool creates the file Validated to confirm that a segment has been tested for DII compliance. Subsequent tools in the development, integration, and installation process use this file to determine whether a segment has been altered, thus indicating that the segment needs to be revalidated.

The following information is captured:

- the version of VerifySeg used to validate the segment
- the date and time validation was performed
- who performed the validation
- a count of all errors and warnings produced by VerifySeg for the segment
- a checksum computed to enable detection of modifications made after the segment was validated.

[45] This represents a change from the previous *I&RTS*. It has been added to correct configuration management problems related to mismatched parent/child segments within an aggregate. To preserve backwards compatibility, VerifySeg will presently generate a warning message if the version number is not specified. However, in a future release it will generate a fatal error so developers should begin to use the new format given here. If the version number is specified, VerifySeg will generate a fatal error if the version number is less than 4 digits or does not match the child's version number.

5.5.1.12 VERSION

The format of the VERSION descriptor is

```
version #:date[:time]
```

where *version #* is the version number for the segment, *date* is the version date (in mm/dd/yyyy format), and *time* is an optional time stamp (in the format hh:mm). Version numbers must adhere to the rules defined in Chapter 3.

> **Note:** This release of the *I&RTS* extends the year from 2 digits to 4 digits to avoid complications when the year 2000 arrives. VerifySeg will issue a warning for any segment that uses less than 4 digits, but since this date is used for documentation purposes only, there is no operational impact if only 2 digits are used.

5.5.2 SegInfo Descriptor Sections

This subsection describes all the segment descriptors that are sections within the SegInfo file.

5.5.2.1 AcctGroup

Syntax for the AcctGroup descriptor is

```
group name:group ID:shell:profile flag:home dir:default profile name
```

where

group name is an alphanumeric string used to identify this account group. The account group name must be unique (i.e., no other account group may have the same name).

group id is a UNIX group id to be inserted into the password file for accounts created from this group. The user id is calculated automatically by examining the password file for user accounts within the same group and then adding 1 to the highest user id. Group ids less than 100 should be avoided.

shell is the UNIX shell to execute when logging in (e.g., /bin/csh, /bin/sh). This parameter should be left blank for NT platforms.

profile flag is 0 if no profiles are allowed, otherwise 1.

home dir is the home directory for the given account group (e.g., /h/AcctGrps/SecAdm).

default profile name is an alphanumeric string identifying the account group's default profile. This name is ignored unless the profile flag is nonzero.

In effect, AcctGroup is a template of what to enter into the /etc/passwd file for accounts within this group.

Group names and profile names are not case sensitive. The maximum number of characters in a group name, including embedded blanks, is 15. The maximum number of characters in a profile name[46] is 64. The maximum number of characters in the home directory pathname is 256.

If the account group is to have a default profile, the installation software will automatically create the profile with the name specified. The profile will be set up to have a classification level of TOP SECRET (unless the segment specifies otherwise), all possible object permissions enabled (see the `Permissions` descriptor), and all possible menu and icon entries enabled. Note that site administrators will not normally assign the default profile to any user because it would provide greater access than is warranted either from a "need to know" perspective, or from a perspective of overwhelming the operator with too many features. The default profile is provided only as a convenient template for creating user profiles.

The profile classification can be explicitly stated by including a line of the form

```
$CLASSIF:classification
```

within the segment descriptor section. Valid classification values are

UNCLASS
CONFIDENTIAL
SECRET
TOP SECRET

5.5.2.2 AppPaths (NT Only)

The `AppPaths` segment descriptor is used to add a list of executables and DLLs to the NT search path. The executables are listed immediately after the segment descriptor as in

```
[AppPaths]
app1.exe
app2.exe
app3.DLL
```

The executables and DLLs must be in the segment's `bin` subdirectory.

The installation tools remove the named executables and DLLs from the NT search path when the segment is deleted. Refer to subsection 5.5.2.25 for more information on shared files.

> **Note:** As with UNIX, it is a violation of the COE to use this technique to insert the current working directory into the NT search path.

5.5.2.3 COEServices

Segments frequently require changes to services provided by the operating system. Make such requests through the `COEServices` descriptor to ensure proper coordination with other segments. One or more entries may follow each keyword.

`$GROUPS` (UNIX only)

Segments may add entries to the `/etc/group` file as follows:

[46] The maximum in the previous *I&RTS* was limited to 15 characters. This has been extended to support those services which describe profiles based on a combination of duty position and organization, or similar approach.

```
$GROUPS
name:group id
```

where *name* and *group id* have the meaning defined by the UNIX `group` file. If the specified name already exists in the group file but with a different group id, an error will be generated.

`$PASSWORDS` (UNIX only)

Segments may occasionally need to add entries into the UNIX password file to establish file ownership. The syntax is:

```
login name:user id:group id:comment:home dir:shell
```

where these entries correspond to the entries in the UNIX `passwd` file. Multiple lines may be included to add multiple password entries.

The installation software inserts an "*" for the password field to ensure that these are system accounts, not actual user login accounts. Segments that need to add a user account must be approved in advance by the Chief Engineer, and then will generally be approved only for COE-component segments.

The installation software processes the `$PASSWORDS` keyword *before* the segment is actually loaded onto disk so that `PostInstall` scripts which need to set file ownership will work properly.

`$SERVICES`

Ports are added to the `/etc/services` (or NT equivalent) system file through the `$SERVICES` keyword. The syntax is:

```
$SERVICES[:comment]
name:port:protocol{:alias}
```

where

name is the name of the socket to add,

port is the port number requested, and

protocol is either `tcp` or `udp`.

The optional *comment*, if provided, will be inserted into the `/etc/services` file by the installation software.

If the port number requested is already in use under another name, an error will be generated. Note that port numbers in the range 2000-2999 are reserved for COE component segments and may not be used by mission application segments.

This keyword should not be necessary for most DCE applications because endpoints are defined dynamically.

5.5.2.4 Community

Many of the descriptor files direct the installation software to insert, delete, replace or otherwise alter blocks of text in ASCII files. The `Community` descriptor is provided to issue similar commands to the installation software for which no corresponding descriptor exists. It is intended to be a "catch all" and

should be used carefully, and only when there is no other way to accomplish the modifications required. `VerifySeg` will fail any segment which attempts to use a `Community` descriptor to modify a file that is already handled by another descriptor. For example, inserting a port entry into `/etc/services` is handled by the `COEServices` descriptor so `VerifySeg` will fail a segment that attempts to do this through a `Community` descriptor.

Segment developers shall use the `Comm.deinstall` descriptor to undo changes made by the `Community` file. `Comm.deinstall` is invoked when a segment is removed and is the inverse of the `Community` file. The `Comm.deinstall` is neither required nor useful if the segment is a permanent segment.

The commands listed below are available for both the `Community` and `Comm.deinstall` files. Blocks of text are delimited by braces, where the opening and closing brace are on a line by themselves. When commands require that a textual search be done, embedded spaces and control characters are ignored during the search.

To illustrate how the commands work, assume the file `IDE.TEST` contains the following text:

```
# Sample file

# Define runtime vars
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data

# set a test var
setenv testvar $HOME

set filec

setenv testvar2 $HOME/data

# end of example file
```

`$APPEND`

Append the block of text which follows to the end of the file.

Example:

```
$APPEND
{
# This is an example to append at the end of a file
source my_script
#
}
```

`$COMMENT:char`

Using the character specified, find the block of text which follows and comment it out. This effectively deletes text, but has the advantage that it can easily be uncommented.

The command sequence

```
$COMMENT:#
{
# set a test var
```

```
setenv testvar $HOME
set filec
}
```

will replace the text to modify the file as follows:

```
# Sample file

# Define runtime vars
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data

## set a test var
#setenv testvar $HOME
#
#set filec

setenv testvar2 $HOME/data

# end of example file
```

Notice that the blank line between `setenv` and `set` is ignored in searching for the lines to delete, but is preserved in the commented out version of the file.

> **Note:** Be careful to note that the '#' character is not a valid comment delimiter for all community files! (e.g., X and Motif resource files use '!' as a comment delimiter.)

`$DELETE [ALL]`

Find the block of text which follows and delete it from the file. If `ALL` is specified, delete every occurrence in the file.

The command sequence

```
$DELETE
{
# set a test var
setenv testvar $HOME
set filec
}
```

will delete the block of text to modify the file as follows:

```
# Sample file

# Define runtime vars
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data


setenv testvar2 $HOME/data

# end of example file
```

Notice that the blank line between `setenv` and `set` is ignored in searching for the lines to delete, but is deleted in the resulting version of the file.

`$FILE:filename`

Name the file to which the commands that follow apply.

Example:

```
$FILE:/h/IDE/Scripts/IDE.JMCIS
```

`$INSERT [ALL]`

Find the first occurrence of the first block of text, then insert the second block of text immediately after it. If `ALL` is specified, insert the second block of text after every occurrence.

Example:

```
$INSERT
{
setenv OPT_DATA $OPT_HOME/data
}
{
setenv OPT_BIN $OPT_HOME/bin
setenv OPT_SRC $OPT_HOME/src
}
```

The resulting changes to the example file are:

```
# Sample file

# Define runtime vars
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data
setenv OPT_BIN $OPT_HOME/bin
setenv OPT_SRC $OPT_HOME/src

# set a test var
setenv testvar $HOME

set filec

setenv testvar2 $HOME/data

# end of example file
```

`$REPLACE [ALL]`

Replace the first occurrence of the first block of text, if found, with the second. If `ALL` is specified, replace every occurrence.

Example:

```
$REPLACE
{
setenv OPT_HOME /h/OPT
```

```
}
{
setenv OPT_HOME /home2/OPT
}
```

Embedded spaces and control characters are ignored in the search, but are preserved in the replacement. Case is preserved in the search and in the replacement.

$SUBSTR:DELETE [ALL] | INSERT [ALL] | REPLACE [ALL]

When performing a textual search, search for a matching substring instead. Insertions, deletions, or replacements are made as indicated.

$UNCOMMENT:char

Find the block of text which follows and uncomment it. The comment character is *char*, but the block of text which follows the `$UNCOMMENT` command does not contain the comment character.

Example (undo the effects of the `$COMMENT` example above):

```
$UNCOMMENT:#
{
# set a test var
setenv testvar $HOME
set filec
}
```

Blank lines will also be uncommented if there are any between

```
# set a test var
```

and

```
set filec
```

Consider a more complete example. The following will insert two new environment variables at the end of the file, replace `OPT_HOME` with `OPTION_HOME`, replace `OPT_DATA` with `OPTION_DATA`, and replace all occurrences of the substring "`stvar`" with "`st_var`". This example also shows the use of comments.

```
/* This is a multi-line comment
 just like in standard C.
*/
# This is a single line comment

# Assume file is in IDE Scripts subdirectory
$FILE:/h/IDE/Scripts/IDE.TEST

$REPLACE
{
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data
}
{
setenv OPTION_HOME /h/OPTION
setenv OPTION_DATA $OPTION_HOME/data
}

$SUBSTR:REPLACE ALL
{
stvar
}
{
st_var
}

$APPEND
{
#------------------------
# BEGIN xxx modifications
#------------------------

setenv my_var /h/IDE

#------------------------
# END xxx modifications
#------------------------
}
```

The resulting file `IDE.TEST` is

```
# Sample file

# Define runtime vars
setenv OPTION_HOME /h/OPTION
setenv OPTION_DATA $OPTION_HOME/data

# set a test var
setenv test_var $HOME

set filec

setenv test_var2 $HOME/data

# end of example file
#-----------------------
# BEGIN xxx modifications
#-----------------------

setenv my_var /h/IDE

#-----------------------
# END xxx modifications
#-----------------------
```

This example shows the use of comments to enclose modifications between a BEGIN/END pair. This technique is recommended when making modifications to community files to make it easier to determine changes made as segments are installed.

> **Note:** This technique is used by the installation software as environment extension files are modified. Therefore, developers must *not* put such comments in environment extension files.

5.5.2.5 Comm.deinstall

`Comm.deinstall` is the inverse of `Community`. Its purpose is to undo modifications made to community files when a segment is removed from the system.

The corresponding `Comm.deinstall` file to undo the changes made in the example from the `Community` subsection is:

```
$FILE:/h/IDE/Scripts/IDE.TEST
$REPLACE
{
setenv OPTION_HOME /h/OPTION
setenv OPTION_DATA $OPTION_HOME/data
}
{
setenv OPT_HOME /h/OPT
setenv OPT_DATA $OPT_HOME/data
}

$SUBSTR:REPLACE ALL
{
st_var
}
{
stvar
}

$DELETE
{
#------------------------
# BEGIN xxx modifications
#------------------------

setenv my_var /h/IDE

#------------------------
# END xxx modifications
#------------------------
}
```

5.5.2.6 Compat

Subsequent releases of a segment are not always backwards compatible. The `Compat` descriptor is used to indicate the degree to which backward compatibility is preserved with the newly released segment. This is achieved by listing version numbers for previous releases which the current release supports. In the sense used here, backwards compatibility means that the segment being released will work with other segments that have been compiled and linked with an earlier release version.

The format of the `Compat` descriptor is a single line containing one of three possible entries:

+ALL	This indicates that the current release is backwards compatible with all previous releases.
-NONE	This indicates that the current release is not backwards compatible with any previous release.
version list	This indicates that the current release is backwards compatible to a list of versions. Version lists are denoted by the `$LIST`, `$EARLIEST`, and `$EXCEPTIONS` keywords.

For example, suppose the new `MySeg` release is version 3.2.5.4 and that it is compatible with all versions from 2.9.1 up to the present with the exception of versions 3.0.1.2 and the 3.1 version series. Then the `Compat` file would contain the following entries:

```
# First number listed is earliest compatible version
$EARLIEST
2.9.1
# Remaining version numbers are exceptions
$EXCEPTIONS
3.0.1.2
3.1
```

When a digit is omitted from the version number, or an asterisk is in place of the digit, there is an assumed wildcard in that digit position. That is, any digits would be acceptable in that position.

The `$LIST` keyword is used to indicate an explicit list of compatible versions. `$LIST` is mutually exclusive with the `$EARLIEST`/`$EXCEPTIONS` keyword pair. When specifying a list, a range can be indicated by the optional keyword `$TO`. Thus, the previous example could also have been done as

```
$LIST
2.9.1 $TO 3.0.1.1
3.0.1.3 $TO 3.0.9
3.2.0 $TO 3.2.5
```

In some cases, one or more patches must be applied to preserve compatibility. The patches are listed by number immediately after the version number by using a colon between patch numbers. This may be done only with the `$LIST` keyword. For example,

```
$LIST
2.9.1:P4:P5
3.0.1.1
3.0.2:P8 $TO 3.0.4:P7
```

This means that the current version is backwards compatible with

- 2.9.1, but only if patches P4 and P5 have been applied
- 3.0.1.1 with no restrictions regarding patches
- 3.0.2 through 3.0.4 with the restriction that patch P8 must be applied to version 3.0.2 and patch P7 must be applied to version 3.0.4.

If no `Compat` file exists, the present version is assumed to not be backwards compatible with any previous releases. That is, `-NONE` is assumed.

5.5.2.7 Conflicts

Two segments may conflict with one another so that one or the other, but not both, can be installed. The `Conflicts` descriptor is used to specify such inter-segment conflicts. The format is a list of conflicting segments in the form:

```
segment name:prefix:home dir[:version{:patch}]
```

where *segment name* is the name of the conflicting segment as given in the segment's `SegName` descriptor file, *prefix* is the conflicting segment's segment prefix, and *home dir* is the conflicting segment's home directory.

The `Conflicts` descriptor is essentially the inverse of the `Requires` descriptor.

5.5.2.8 Data

The `Data` descriptor is used to describe where data files are to be logically loaded and their scope (global, local, or segment). Only one of the three scopes may be specified in the descriptor; that is, a data segment has one and only one scope.

The syntax is

```
$SEGMENT:segname:prefix:home dir
```

for segment data, or

```
$LOCAL:segname:prefix:home dir
```

for local data, or

```
$GLOBAL:segname:prefix:home dir
```

for global data, where *segname*, *prefix*, and *home dir* refer to the affected segment. The *segname* and *prefix* must match the name given in the affected segment's `SegName` descriptor. Figure 5-9 shows that the data to install is underneath the segment's `data` subdirectory.

5.5.2.9 Database

The `Database` segment descriptor is used to identify information such as object dependencies that are within the database and therefore cannot be resolved without the use of the DBMS. There are five keywords used under this descriptor to track object-level information: `$REFERENCES`, `$MODIFIES`, `$ROLES`, `$SCOPE`, and `$ACCESSES`. The first four are used by database segments, the last is used by database application segments. Their usage is discussed below.

`$SCOPE:`*`scope`*

This keyword specifies the scope of the database objects. Legal values for *scope* are `UNIQUE`, `SHARED`, and `UNIVERSAL`. Scope is required for database segments, but it is not presently used. It is reserved for future use and required now so that segments will not require modifications later.

`$REFERENCES`

The `$REFERENCES` keyword is followed by a list of the individual database objects that the database segment depends upon which are external to the segment. The `Requires` segment descriptor must be used to state a dependency upon the segments whose objects are listed under `$REFERENCES`. Version compatibility will be checked using the `Requires` descriptor so it is not repeated here. The format for the object list is

```
$REFERENCES
object name:schema
```

For example, assume that the GSORTS database segment references the `COUNTRY_CODE` table in the S&M segment and the `PORTS` table in the NID segment. The schema owners for S&M and NID respectively are `TABLE_MASTER` and `NID`. The appropriate descriptor is

```
$REFERENCES
COUNTRY_CODE:TABLE_MASTER
PORTS:NID
```

$MODIFIES

The $MODIFIES keyword is followed by a list of the external database objects that the database segment modifies by adding triggers, or by including them in procedures or functions. All segments whose objects are listed here must also appear under the Requires descriptor. The format for the object list is

```
$MODIFIES
object name:schema:modification type:modification name
```

The *object name* and *schema* follow the same rules as the $REFERENCES keyword. *Modification type* is used to stipulate what has been done. Its legal values are TRIGGER for database triggers or PROCEDURE for database functions, procedures, or packages. *Modification name* is the name of the trigger or procedure that is attached to the object. An example follows defining a trigger named GSORTS_NID_COPY that is attached to the NID database's PORTS table.

```
$MODIFIES
PORTS:NID:TRIGGER:GSORTS_NID_COPY
```

$ROLES

The $ROLES keyword is followed by a list of the database roles created by the database segment. Its format is

```
$ROLES
role name
```

An example that defines two roles follows.

```
$ROLES
EWIR_RO
EWIR_DATA1_RW
```

It is recommended that comments be placed in the segment descriptor to describe what these roles are for and how they are intended to be used. This is a convenient place to document such important information.

$ACCESSES

The $ACCESSES keyword is used in a software segment rather than a database segment. It associates individual applications within a software segment to their supporting database roles. Its format is

```
$ACCESSES
application name:role name:segment name
```

The *application name* is the name of the executable within the segment. *Role name* is the name of the database role used by the application. *segment name* is the name of the database segment that owns that role. That segment will be searched by the installer tool, if necessary, to obtain the DBO account name. An example follows associating the EWIR_WIDE application to the EWIR_RO role.

```
$ACCESSES
EWIR_WIDE.FMX:EWIR_RO:EWIRDB
```

Note: Do not confuse the Database segment *descriptor* with the database segment *type*. The segment descriptor, described in this

subsection, describes specialized processing for the COE to perform on a segment which is of segment type 'database.'

5.5.2.10 DCEClientDef

This segment descriptor is used to define the characteristics of DCE Clients. The server installation script reads the `DCEClientDef` section from the `SegInfo` file for installation specific information. The associated keywords are used to describe the DCE client.

Table 5-6 lists the keywords applicable to DCE segments that use the DCE COE application development library. As indicated in the table, some are for servers only, some are for clients only, and some may be used for both client and server segments. For a more complete description of these keywords and the use of the DCE COE library please refer to the *DII COE DCE Programmers Guide*.

Keyword	Client	Server
ACLMGRDEFAULT	n/a	M
ACLMGRINFO	n/a	O
ACLMGRTYPE	n/a	O
ACLMGRUUID	n/a	O
ATTRIBUTE	O	O
AUDITINFO	n/a	O
DCEACL	n/a	*
DCEADMINGROUP	n/a	O
DCEBOOT	n/a	O
DCECLIENT	M	n/a
DCEGROUP	O	O
DCESERVICE	n/a	M
DEBUGMESSAGES	n/a	O
DFSFILES	O	O
INTERFACE	M	M
MESSAGES	n/a	O
MGMTMAPPING	n/a	O
OBJUUID	n/a	M
PERMISSION	n/a	M
RPCSECURITY	n/a	O
SERVERTHREADS	n/a	O
SERVICEABILITY	n/a	O
UUID	n/a	M

Legend: **M - Mandatory** **O - Optional** **n/a - Not Applicable** *** - Reserved for Future Use**

Table 5-6: DCE Client and Server Keywords

`$ATTRIBUTE`

The format for this keyword is the same for both clients and servers. Refer to subsection 5.5.2.11 for a full description.

`$DCECLIENT client:title`

client is the name of the client application and *title* is a brief description of the client application. DCE client segments require the `$DCECLIENT` keyword. This provides the name of the client application and annotation.

Example:

```
$DCECLIENT CALCclient:Basic calculator client
```

`$DCEGROUP`

The format for this keyword is the same for both clients and servers. Refer to subsection 5.5.2.11 for a full description.

`$DFSFILES`

The format for this keyword is the same for both clients and servers. Refer to subsection 5.5.2.11 for a full description.

`$INTERFACE client:server:CDS entry`

The `$INTERFACE` keyword identifies the name of the server and the location of the `rpcprofile` used to initiate servers. *client* is the name of the client application, *server* is the identity of a server used by the client, and *CDS entry* is the location in the Cell Directory Service (CDS) of an `rpcgroup` or `rpcprofile` used to initiate a search for servers. A client may make use of multiple servers, including servers offered by other segments.

Example:

```
$INTERFACE CALCclient:CALCserver:/.:/h/CALC/groups/servergroup
```

Note: Segments which use the `DCEClientDef` descriptor must also indicate the `DCE` segment attribute or else the COE tools will issue a fatal error.

5.5.2.11 DCEServerDef (UNIX Only)

This segment descriptor is used to define characteristics of DCE servers. It is not required, nor is it legal, for DCE client applications. The associated keywords are used to describe the server. The server installation script reads the `DCEServerDef` section from the `SegInfo` file for installation specific information. Table 5-6 lists the applicable keywords for describing DCE servers. Note that some of the keywords are also used for describing characteristics of client segments.

Most of these keywords are used by the standard DCE installation program to set attributes in CDS, to include attributes within the configuration entry for the application. Refer to the *DII COE DCE Programmer's Guide* for more information.

Before describing the applicable keywords, there are some important things to note about DCE servers.

- Use `$DCESERVICE` instead of the `$SERVERS` keyword (`Network` descriptor) to define DCE-based servers.
- Document Distributed File Service (DFS) files with the `$DFSFILES` keyword.
- Include a `$PASSWORDS` entry in `COEServices` to establish a UNIX userid for each server principal.

- Developers should normally provide a single DCE server in a segment. It would be unusual to need to provide more than one.

> **Note:** Segments which use the `DCEServerDef` descriptor must also indicate the `DCE` segment attribute or else the COE tools will issue a fatal error.

`$ACLMGRDEFAULT service:interface:type:name:permissions`

Values of the AclMgrDefault attribute are used to give the server ACL an initial set of values. This attribute is multi-valued and can contain any combination of 'group' or 'user' ACL entries. The meaning of the parameters are:

- *service* – The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *interface* – The name of an interface implemented by the server. This interface must match the interface name defined in an IDL file and as defined in the `$INTERFACE` keyword.
- *type* - one of the following values:
  - `USER`
  - `GROUP`
  - `ANY_OTHER`
  - `UNAUTHENTICATED`
- *name* - Used with `USER` or `GROUP` to identify the specific user or groups.
- *permissions* - This field is defined in the `$PERMISSION` keyword. The values used are defined in the name field.

Following are examples of the `$ACLMGRDEFAULT` keyword:

```
$ACLMGRDEFAULT CALCserver:calculator:GROUP:CALC-users:ast
$ACLMGRDEFAULT CALCserver:calculator:UNAUTHENTICATED:t
```

`$ACLMGRINFO service:mgr_name:desc`

This keyword provides ACL management information. The parameters are:

- *service* - The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *mgr_name* - The ACL manager name.
- *desc* - A description (annotation) of the Reference Monitor.

The following is an example:

```
$ACLMGRINFO CALCserver:calculators:Sample Calculator Refmon
```

If this keyword is omitted, the ACL manager is given the same name as the server application (e.g., CALCserver).

`$ACLMGRTYPE service:obj_type:structure_type`

This keyword is reserved to define the structure and type of the data file used to support the standard ACL Manager. It can contain one or more of the supported object types and one of the structure types. The meaning for each parameter follows.

- *service* - The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *obj_type* - The following object types have been defined:
  - `aclobject` - supports ACLs on simple objects
  - `defobject` - supports default inheritance ACLs on objects
  - `defcontainer` - supports default inheritance ACLs on containers

If the keyword is omitted, the default is `aclobject`.

- *structure_type* - The following structural attributes are defined:
  - `flat` - the database contains no hierarchical structure
  - `hier` - the database supports full hierarchy (e.g. a filesystem)
  - `bilevel` - the database does not support containers within containers
  - `sparse` - the database supports sparse searching
  - `noleaf` - the database permits hierarchy but only as a side effect of creating a leaf

  If the keyword is omitted, the default is `flat`.

> **Note:** The initial release supports only `flat`, `bilevel`, and `hier`.

The following is an example:

```
$ACLMGRTYPE CALCserver:aclobject:flat
```

`$ACLMGRUUID service:uuid`

Every ACL manager defines a UUID that represents a set of permissions supported by the ACL manager. This keyword allows the user to define this UUID. The parameters are:

- *service* - The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *uuid* - The combined major and minor version numbers identify one generation of an interface

If the keyword is omitted, a new unique ID is automatically generated.

The following is an example:

```
$ACLMGRUUID CALCserver:6ba40bf6-e2ee-11cf-8d13-ce9cdd02aa77
```

`$ATTRIBUTE name:[uuid]:multivalued:encoding:annotation`

The DCE COE library makes use of pre-defined attributes within the CDS configuration entry for an application. The application can define additional attributes by using the `$ATTRIBUTE` keyword. The COE installation process uses this keyword to define the attribute in the CDS schema.

Each attribute type definition in the schema consists of attribute type identifiers (UUID and name) and semantics that control the instances of attributes of this type. An attribute instance is an attribute that is attached to an object and has a value (as opposed to an attribute type, which has no values but simply defines the semantics to which attribute instances of that attribute type must adhere). Attribute instances contain the UUID of their attribute type.

The identifiers of attribute types are a name and a UUID. Generally, the name is used for interactive access and the UUID for programmatic access. The client can also have `$ATTRIBUTE` entries so take care not to confuse the two.

The meaning of each parameter follows:

- *name* - The name of the attribute.
- *uuid* - The UUID of the attribute.
- *multivalued* - Legal values are `yes` or `no`. The *multivalued* flag specifies whether or not multiple instances of the attribute can be attached to a single application. For example, if the multivalued flag is set `yes`, a single application can have multiple instance of attribute Type A. If the flag is set to `no`, a single application an have only one instance of attribute Type A.
- *encoding* - This defines the legal encoding for instances of the attribute type. The encoding controls the format of the attribute instance values, such as whether the attribute value is an integer, string, a UUID, or a vector of UUIDs that define an attribute set. Legal values for this parameter are: `any`, `void`, `printstring`, `stringarray`, `integer`, `byte`, `uuid`, `i18n_data`, `attrset`, and `binding`.
- *annotation* - The annotation field is text that describes the function of the attribute.

The following is an example (this is intended to be a single line):

```
$ATTRIBUTE unknown_intercell_comms:171e0ff2c-d12e-11de-dd7b-
080009353559:no:integer:Handles intercell access control for foreign
users
```

`$AUDITINFO service:first:num_events:msg code`

This keyword establishes the audit event numbering and message code capability. The parameters are:

- *service* - The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *first* - The first number of the audit event.
- *num_events* - The number of events.
- *msg code* - 3-character message component for events (see the `$SERVICEABILITY` keyword)

The following is an example:

```
$AUDITINFO CALCserver:281587713:2:CAL
```

`$DCEADMINGROUP groupname`

Members of this group are used to control administrative access to application information. These members are able to change acl's, add members to groups, start/stop servers, install/deinstall clients and servers.

- *groupname* - The administrative group name is normally composed of the segment prefix and the word "`admin`." Therefore if the segment prefix is `CALC`, the default group name for administration is `CALC-admin`. The default setting is `SEGMENT-admin`.

The following is an example:

```
$DCEADMINGROUP CALC-admin
```

$DCEBOOT service:starton

The $DCEBOOT attribute identifies when a server should be started. The value is a list of one or more of the following which may not be modified after creation.

- *service* - The name of the server application. This is the same value as found in the $DCESERVICE service field.
- *starton* - One or more of the keywords can be used but must be separated by semicolons.
    - auto - Start if a remote call that would be serviced by this server is received by dced. This is ignored for those servers that are repositories.
    - boot - Start at system startup.
    - explicit - Start if dced receives a command to start the server (such as the server start command in dcecp).
    - failure - Start if dced detects that the server exited with a non-successful error code.

Following are several examples of the $DCEBOOT keyword:

```
$DCEBOOT CALCserver:boot;explicit;failure
```

This example states that the CALCserver is started at boot time. If the server exits with a non-successful error code it will automatically be restarted. The server can also be started from the command line.

```
$DCEBOOT CALCserver:boot;failure
```

This examples shows the CALCserver starting only at boot time and when a error has occurred.

$DCEGROUP groupname

Additional groups may be needed for specific applications. For example a CALC-adders group might be created for a calculator application containing users who are allowed to perform the add operation but not the subtract, division or multiplication functions.

- *groupname* - The name of a user group used to control access to the server services. The group *servername*-users is automatically created and does not require a $DCEGROUP entry..

The following is an example:

```
$DCEGROUP CALC-adders
```

$DCESERVICE service:UNIXid[:principal[:group[:org]]]

DCE server segments require the $DCESERVICE keyword. This provides the name of the server application, ownership and run time authentication principle. The applicable parameters are:

- *service* – The name of the server application. A segment may contain multiple servers. When there is only one server in a segment, the name should be *SegPref*server where *SegPref* is the segment prefix. When there are multiple servers in the segment, each one is identified by a separate $DECSERVICE entry and should be uniquely named using the segment prefix.
- *UNIXid* – The UNIX account used in running the server. Usually supplied by a separate $PASSWORD keyword.
- *principal* – The name for the DCE principle to use in running the server. Default is the same as the server.
- *group* – The group used to control access to server CDS entries. Each server principal belongs to this group. Default is SEGMENT-servers.

- *organization* – The DCE organization for the server principal accounts. Default is none.

The following is an example of a $DCESERVICE entry with the minimum required parameters:

```
$DCESERVICE CALCserver:CALC
```

Note: The UNIX account must exist before the segment is installed. Otherwise the installation will be unsuccessful.

The following is an example which uses all the parameters:

```
$DCESERVICE CALCserver:CALC:CALC:engineering:acom
```

In this example the install script will create the DCE account CALC, the group engineering and the organization acom, if they do not already exist. If these fields are blank the principal used in running the server is CALCserver, the group is none, and the organization is none.

$DEBUGMESSAGES service:routing

service is the name of the service (e.g., CALCserver) and *routing* specifies how and where the debug message should be sent. The format for *routing* is:

```
component:sub_comp.level,...:out_form:dest[;out_form:dest...]
[GOESTO:{sev | component}]
```

where *out_form*, *dest*, and *sev* have the same meanings as for the $MESSAGES keyword. *component* is the three-character serviceability component code for the program whose debug message levels are being specified, *sub_comp.level* is a serviceability subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Nine serviceability debug message levels (specified respectively by single digits from 1 to 9) are available. The precise meaning of each level varies with the application or DCE component in question, but the general notion is that ascending to a higher level (for example, from 2 to 3) increases the level of informational detail in the messages. Setting debug messaging at a certain level means that all levels up to and including the specified level are enabled.

Note: Multiple subcomponent/level pairs can be specified. If there are multiple subcomponents and it is desired to set the debug level to be the same for all of them, then the form: component:*.level will do this (where the * is used as a wildcard to specify all subcomponents).

The following are examples of $DEBUGMESSAGES:

```
$DEBUGMESSAGES CALCserver:coe:*.9:STDOUT:-
$DEBUGMESSAGES CALCserver:coe:*.4:TEXTFILE:/tmp/log_%ld;STDERR:-
```

$DFSFiles

This keyword is similar in purpose to the FilesList segment descriptor (subsection 5.5.2.13). It is used instead of FilesList because the files listed are maintained by DFS, not by the native operating system. The keyword is followed by a list of filenames in the form:

```
filename access
```

where *filename* is the DFS filename used by the application, and *access* indicates the operations performed on the file (RWX). All file names shall start with /.../cellname/fs/.

This keyword is provided for information only.

$INTERFACE service:interface:title

The $INTERFACE keyword defines the server interface as presented in the IDL file. This information **must** match exactly. The $INTERFACE keyword describes a set of runtime routines that allows a client program to use a particular service provided by another application program. The parameters are:

- *service* – A service entry (server application) from the $DCESERVICE keyword.
- *interface* – The name of an interface implemented by the server. This interface must match the interface name defined in an IDL file.
- *title* - The title of the interface, used as an annotation in the DCE endpoint map.

Example:

```
$INTERFACE CALCserver:calculator:Basic Sample calculator Application
```

$MGMTMAPPING service[:string]

This keyword is used to control and configure the management functions that all DCE applications support. Management functions allow a client to request interface information, server principal name, or statistics from the server, to ping the server, or to stop the server. There are five management operations that define the relationship between permissions understood by the ACL manager/Reference monitor permissions. This keyword defines the permissions that must be present to allow the client to perform the management function. The ACL to be checked is attached to the srvrexec object for the server.

The parameters are:

- *service* - The name of the server application. This is the same value as found in the $DCESERVICE service field.
- *string* - The permissions to allow the client to perform management function. If the $MGMTMAPPING keyword is not specified or this parameter is omitted, tttc is assumed which represents the standard 'test' and 'control' permissions.

The following is an example:

```
$MGMTMAPPING CALCserver:ttttc
```

$MESSAGES service:routing

The $MESSAGES and $DEBUGMESSAGES keywords are used to set DCE serviceability options. The parameters are:

- *service* - is the name of the service (Same as in $DCESERVICE)
- *routing* - how to route messages to their destination. This parameter is of the form

```
sev:out_form:dest[;out_form:dest . . . ] [GOESTO:{sev | comp}]
```

where

- *sev* - Specifies the severity level of the message, and must one of the following: FATAL, ERROR, WARNING, NOTICE, or NOTICE_VERBOSE. If the message is to apply to all severity levels, use the wildcard character * as the severity level value.
- *out_form* - Specifies how (e.g., output form) the messages of a given severity level should be processed. The legal values are BINFILE, TEXTFILE, FILE, DISCARD, STDOUT, or STDERR. *out_form* may be followed by a two-number specifier of the form: *.gens.count* where *gens* is an integer that specifies the number of files (i.e., generations) that should be kept and *count* is an integer specifying how many entries (i.e., messages) should be written to each file. The wildcard character * may be used for *gens* or *count* to indicate an unlimited number of generations or messages respectively.
- *dest* - Specifies where (e.g., destination) the message should be sent and is a pathname. Filenames may not contain colons or periods. The field can be left blank if the *out_form* specified is DISCARD, STDOUT, or STDERR. The field can also contain the C formatting string %ld in the filename which, when the file is written, will be replaced by the process ID of the program that wrote the message. Multiple routings for the same severity level can be specified by adding the additional desired routings as semicolon-separated strings in the following format:
  NOTICE:BINFILE.50.100:/tmp/log%ld;STDERR:-
- GOESTO - Permits messages for the severity whose routing specification it appears in to be routed to the same destination as those for the other specified severity level. Examples are:
  WARNING:STDERR:GOESTO:FATAL
  FATAL:STDERR:;FILE:/tmp/foo
  This means that WARNING messages should show up in three places: twice to stderr, and then once to the file /tmp/foo.

The following is an example of the $MESSAGES keyword:

```
$MESSAGES CALCserver:*:STDOUT:-
```

$OBJUUID service:interface:objuuid

Standard DCE has the ability for servers to associate themselves with “objects” (identified by uuid’s), and for clients to request a binding to any server providing a specified object. The objects supported by a server are identified within its rpcentry within CDS. This facility is designed to allow the location of coarse-grained objects (e.g. specific branches of a bank, or classes of users). It is not designed for fine-grained objects (e.g. an individual account in a bank).

The DCE COE library allows the use of this capability. The server is responsible for registering supported objects using standard DCE calls. The client must have the uuid’s of desired objects pre-configured within its **services** attribute for the appropriate interface.

- *service* - name of service (same as that listed in the $DCESERVICE keyword)
- *interface* - name of the interface (same as identified in the $INTERFACE keyword)
- *objuuid* - The universal unique identifier that identifies a particular RPC object. A server specifies a distinct object UUID for each of its RPC objects; to access a particular RPC object, a client uses the object UUID to find the server that offers the object.
    - Sometime the object UUID is the “nil” UUID; when calling an RPC runtime routine, you can represent the nil UUID by specifying NULL. In this case, the object UUID does not represent any object.

The following is an example of the $OBJUUID keyword:

```
$OBJUUID CALCserver:calculator:01eb03d6-0688-1acb-97ad-08002b12b8f8
```

$PERMISSION service:interface:permission:name:value

The $PERMISSION keyword is used to define a set of access controls to maintain control over the interface. There are several ACL bit permissions that are recommended by OSF, listed in Table 5-7. Additional powers of 2 may be used for application-specific permissions. In the examples, values 128 and 256 are extensions specific to the CALC example. These values provide ACL management for the add and subtract interface.

Permission	Name	Value
r	read	1
w	write	2
e	execute	4
c	control	8
i	insert	16
d	delete	32
t	test	64

Table 5-7: Recommended ACL Bit Permissions

The meaning for each parameter is as follows:

- *service* – A service entry (server application) from the $DCESERVICE keyword
- *interface* – The name of an interface implemented by the server. This interface must match the interface name defined in an IDL file.
- *permission* - A single character value used within ACL permission strings.
- *name* - A short title for the permission, used primarily as a comment.
- *value* - A numeric value for the permission. Must be a power of two. If possible, choose a permission value from Table 5-7 but additional values may be used if necessary. The assignment of different meanings to the values in this table is strongly discouraged.

The following are examples of entries in the SegInfo file:

```
$PERMISSION CALCserver:calculator:c:control:8
$PERMISSION CALCserver:calculator:t:test:64
$PERMISSION CALCserver:calculator:a:add:128
$PERMISSION CALCserver:calculator:s:substract:256
```

$RPCSECURITY service:interface:security

The $RPCSECURITY keyword specifies the protection levels supported. These levels identify how much information in network messages is encrypted.

- *service* - is the name of the service implementing the interface (Same as in $DCESERVICE)
- *interface* - is the name of the interface (Same as in $INTERFACE)

The *security* parameter is composed of several fields:

```
authentication type:[principle name:protection level:authentication
service:authorization service]
```

where

authentication type is one of the following:

- `none` - This type has no further information.
- `dce` - This type is followed by the following fields:
  - *principle name*
  - *protection level* - one of the following values:
    - `default` - Uses the default protection level for the specified authentication service.
    - `none` - There is no protection level.
    - `connect` - Performs authentication only when a client and server establish a relationship (or connection). This level performs an encrypted handshake when the client first communicates with the server. Encryption or decryption is not performed on the data sent between the client and server.
    - `call` - Attaches a verifier to each client call and server response that protects the system -level metadata of every RPC call (but not the application-level data). This level does not apply to remote procedure calls made over a connection-based protocol sequence.
    - `pkt` - Ensures that all data received is from the expected client. This level attaches a verifier to each message.
    - `pktinteg` - In addition to protecting metadata, ensures the integrity of the application-level data (RPC call and return parameters) transferred between two principals, that is, that none of it has been modified in transit.
    - `pktprivacy` - In addition to protecting metadata and integrity, encrypts all application-level data, thus guaranteeing its confidentiality.
  - *authentication service* - one of the following:
    - `default` - DCE default authentication service.
    - `none` - No authentication.
    - `secret` - DCE shared-secret key authentication.
  - *authorization service* - This is the process of checking a client's permissions to an object that is controlled by the server. Access checking is entirely a server responsibility. Possible values are:
    - `default` - No authorization information is provided to the server, usually because the server does not perform access checking.
    - `name` - Only the client principal name is provided to the server. The server can then perform authorization based on the provided name.
    - `dce` - The client's credentials is provided to the server with each remote procedure call that is made using the binding parameter.

Examples of the `$RPCSECURITY` keyword are:

```
$RPCSECURITY CALCserver:calculator:dce:CALCserver:default:default:dce
$RPCSECURITY CALCserver:calculator:dce:CALCserver:pktprivacy:secret:dce
```

`$SERVERTHREADS service:num_threads`

This keyword defines the number of call threads that the DCE runtime creates in order to service incoming RPC requests. Parameters are:

- *service* - The name of the server application. This is the same value as found in the `$DCESERVICE` service field.
- *num_threads* - The number of threads allocated. If not specified the default is 5.

The following is an example:

```
$SERVERTHREADS CALCserver:5
```

$SERVICEABILITY service:code

This keyword identifies the serviceability message code for the application, as defined in the application serviceability messages file. The serviceability messages file defines message text and audit message numbers for use by the application. All serviceability messages contain a six-letter sequence identifying the "technology" and "component" that generated the message.[47] Determine a three-letter lower case component name for the application derived from the segment prefix (e.g., In the example used in this subsection, `CALC` is the segment prefix so the "component" part is `cal`). These three letters will appear on every system-generated message from the application. Insert the component name in the front of the SAMS file, as shown in the sample below. There are no differences in defining a SAMS file for a COE application compared to any other DCE application.

> **Note:** If using the sample application `CALC.sams` file as a template, there are numerous places where the component name is used in variable names by convention, and must be changed for a different application.

```
# Part I
# This part defines the lowest-level table, the one that contains
# all the messages (defined in the third part) in a
# straight array.
component         cal
table             cal_table
technology        dce
```

The DCE COE library functions make use of the OSF DCE 1.1 serviceability interfaces to generate and manage error messages. The server management interface allows messages of different severity to be turned on or off and routed to different locations (e.g. error log, stderr, etc.).

The parameters for this keyword are:

- *service* - is the name of the service (Same as in `$DCESERVICE`)
- *code* – This is a three-letter component used to identify serviceability message files and serviceability messages for this server. It can be a number or lower case text.

The following is an example of a `$SERVICEABILITY` keyword:

```
$SERVICEABILITY CALCserver:cal
```

$UUID service:interface:uuid version

This is the interface UUID. Each DCE interface has a unique identifier (UUID) to ensure compatibility of the client and server. This UUID identifies a specific RPC interface. An interface UUID is declared in an RPC interface definition (an IDL file) and is required element of the interface and the `SegInfo` file.

- *service* – A service entry (server application) from the `$DCESERVICE` entry

[47] Applications are supposed to be identified with the technology **dce** and an identifying number assigned by the OSF. Until a block of numbers are assigned for COE applications, a unique component name derived from the segment prefix should be used.

- *interface* – The name of an interface implemented by the server. This interface must match the interface name defined in an IDL file.
- *uuid version* - The combined major and minor version numbers identify one generation of an interface. Version numbers (1.0) allow multiple versions of an RPC interface to coexist. Strict rules govern valid changes to an interface and determine whether different versions of an interface are compatible. The offered and requested interface are compatible under the following conditions:

    - The interface requested by the client and the interface offered by the server have the same major version number
    - The interface requested by the client has a minor version number less than or equal to that of the interface offered by the server.

An example of the `$UUID` keyword is:

```
$UUID CALCserver:calculator:0073a028-fbdb-1e53-908e-08002b13ca26 1.0
```

5.5.2.12 Direct

The segment descriptor `Direct` allows a segment to issues special instructions to the installation tools. If the segment is part of an aggregate, the directives below apply *only* to the segment in whose `SegDescrip` subdirectory the directives appear.

`$ACCTADD:executable`

This keyword informs the installation software that the specified *executable*, in the segment's `bin` subdirectory, should be run each time a user account is added to the system. `VerifySeg` will flag use of this keyword as a warning to highlight that it is being used. Prior permission must be given by the Chief Engineer before this keyword can be used.

`$ACCTDEL:executable`

This keyword informs the installation software that the specified *executable*, in the segment's `bin` subdirectory, should be run each time a user account is deleted from the system. `VerifySeg` will flag use of this keyword as a warning to highlight that it is being used. For security reasons, prior permission must be given by the Chief Engineer before this keyword can be used.

`$CMDLINE`

Segments which provide a command-line access must insert this keyword in their segment.

`$KEY:`*`request`*`:`*`key`*

Several of the keywords presented here require authorization by the Chief Engineer. Thus, `$KEY` must be provided for each requested permission. *key* is the authorization key provided by the Chief Engineer. *request* is an indication of the type of request being made. Requests are grouped by the type of request being made (e.g., security-related, installation-related) and are one of the following values:

`INSTALL`	for permission to run `PostInstall`, `PreInstall`, and `DEINSTALL` with root permission
`ACCTS`	to use any of the account creation/deletion keywords (e.g., for `$ACCTDEL`, `$ACCTADD`, `$PROFADD`, `$PROFDEL`, and `$PROFSWITCH`)
`CMDLINE`	to use the `$CMDLINE` keyword
`SUPERUSER`	to use the `$SUPERUSER` keyword

A separate authorization key and $KEY entry is required for *each* request group, but the key applies to any and all requests within that group.

$NOCOMPRESS

The MakeInstall tool automatically compresses segments to reduce the amount of space required on disk or tape, and to reduce the download time. The installation tools automatically decompress segments at installation time. The $NOCOMPRESS keyword indicates that compression is *not* to be performed.

$PROFADD:executable

This keyword operates in the same fashion as $ACCTADD, except that it is used when profiles are added to the system.

$PROFDEL:executable

This keyword operates in the same fashion as $ACCTDEL, except that it is used when profiles are added to the system.

$PROFSWITCH:executable

This keyword is similar to $PROFADD except that the executable is run whenever a user currently logged in switches from one profile to another. The executable is *not* run when the user first logs in; it is run only when a profile switch is made.

$READ_ONLY

This keyword informs the installation software that the segment can be run from a read-only medium (e.g., CDROM). This implies that the segment does not modify any files under its installation directory.

$REBOOT

The presence of this keyword indicates that the installation software should automatically reboot the computer after the segment is loaded. If several segments have been selected for loading at one time, the reboot operation will not occur until all segments have been processed. The operator will be notified before the reboot occurs and given the option to override the reboot directive.

$REMOTE[:XTERM | :CHARBIF]

This keyword indicates that the functions (*all* functions) provided by this segment can be executed remotely. At installation time, the installation software will note that this segment can be executed remotely. If the XTERM attribute is present, it indicates that the segment can also be accessed via an "xterm" capability, and output will be routed to the display surface pointed to by the DISPLAY environment variable setting. If the CHARBIF attribute is present, it indicates that the segment supports a character-based interface. CHARBIF and XTERM will normally be mutually exclusive.

By default, segments are assumed to be locally executable only.

$ROOT:PostInstall | PreInstall | DEINSTALL

The presence of this keyword indicates that the specified descriptor must be run with root privileges. A separate $ROOT entry is required for each descriptor. VerifySeg will flag use of this keyword as a warning to highlight that it is being used. For security reasons, prior permission must be given by the Chief Engineer before this keyword can be used. $ROOT requires the $KEY keyword as well.

$SELF_CONTAINED

This keyword informs the installation software that the segment remains in its original condition after installation, with all files intact under the installation directory. It also informs the installation software that any changes made during installation (e.g. in PreInstall and PostInstall) do not have side effects if run multiple times. This allows the installation software to use an installed version of this segment as the source medium for a subsequent installation on another machine.

$SUPERUSER

Segments which provide or require superuser privileges, via a command-line or otherwise, must insert this keyword in their segment. Note that the $KEY keyword must also be used to verify that Chief Engineer approval has been obtained.

$USES_UNINSTALL

This keyword applies to NT segments only. The segment installer software normally handles registration of "uninstall" information for segments. However, some segments, particularly COTS segments, may already do this themselves. In such cases, the segment *must* use the $USES_UNINSTALL keyword to indicate to the segment installer that the segment itself is handling uninstall registration. When this keyword is present, the segment installer does not perform any uninstall registration during installation. This keyword may only be used for COTS segments or as authorized by the Chief Engineer.

5.5.2.13 FilesList

FilesList is a list of files and directories that make up the current segment. It is required for COTS segments. For other segment types, it is useful for documenting community files modified or used by the segment. The reason that this descriptor is required for COTS segments is that COTS products do not usually conform to the DII-mandated directory structure. Therefore, the location of files modified by or contributed by the segment is not usually readily apparent.

FilesList may contain the following keywords:

$DIRS	a list of directories which this segment adds to the system. All files in the directory are assumed to belong to the segment.
$FILES	a list of files which this segment adds to the system.
$PATH	a shortcut for specifying a pathname. Succeeding $DIRS or $FILES are relative with respect to the path specified.

A keyword must precede any list so that it will be clear whether a directory or a file is intended.

As an example, assume a segment to be installed creates the following four subdirectories

```
/h/data/test/data1
/h/data/test/data2
/h/data/opt/data3
```

```
/usr/opt/temp
```

and adds three files (`f1`, `f2`, `f3`) to the `/etc` subdirectory. Then the file `FilesList` could contain the following entries:

```
$PATH:/h/data
$DIRS
test/data1
test/data2
opt/data3
$DIRS
/usr/opt/temp
$PATH:/etc
$FILES
f1
f2
f3
```

The `$DIRS` keyword before `/usr/opt/temp` is not necessary, but is shown to illustrate that `FilesList` may contain multiple occurrences of the keywords.

For COTS products, this descriptor must be used to list:

1. all files and directories the product adds that lie outside the segment's assigned directory, and
2. any community file the COTS product modifies unless the modification is made by the COE installation tools.

For example, assume a COTS segment adds a port to `/etc/services` through the `COEServices` segment descriptor. Further, assume that the vendor provides a program that directly modifies the `/etc/group` file as part of the installation process. Then `FilesList` must list `/etc/group` but does not need to include `/etc/services` because the installation tool modifies it as a result of the `COEServices` descriptor.

5.5.2.14 Hardware

The `Hardware` descriptor defines the computing resources required by the segment. Keywords `$CPU` and `$MEMORY` may appear only once; both are required for all segments, except that `$MEMORY` may be omitted for a data segment. `$DISK` and `$PARTITION` are mutually exclusive, but one must appear in the segment descriptor. `$DISK` may appear only once, but `$PARTITION` may appear multiple times. `$OPSYS` and `$TEMPSPACE` are optional.

`$CPU:platform | ALL`

platform is one of the supported platform constants listed in subsection 5.3 for `MACHINE_CPU`, or `ALL`. If `ALL` is given, it indicates that the segment is hardware independent (e.g., a data segment). If platform is a generic constant (e.g., HP or PC), it applies to all platforms of that class. Thus,

```
$CPU:PC
```

indicates that the software can be loaded on any PC, whether the PC is a 386, 486, or Pentium class machine. However,

```
$CPU:PC386
```

indicates that the software can be loaded on a 386 or better class platform. Similarly, HP712 indicates that the software can be loaded on an HP712 or better class platform that is binary compatible with the HP712.

`$DISK:size[:reserve]`

size is expressed in kilobytes and is the size of the segment, including all of its subdirectories, at install time. The COE tool `CalcSpace` (see Appendix C) will compute the disk space occupied by a segment and update this keyword. *reserve* is also expressed in kilobytes and is the *additional* amount of disk storage to reserve for future segment growth.

`$MEMORY:size`

size is expressed in kilobytes of Random Access Memory (RAM) required.

`$OPSYS:operating system | ALL`

operating system is one of the supported platform constants listed in subsection 5.3 for `MACHINE_OS`, or `ALL`. If `ALL` is given, it indicates that the segment is operating system independent. Dependencies on a particular version of the operating system are defined in the `Requires` descriptor where a dependency on a specific segment (e.g., operating system with a particular version) is described.

`$PARTITION:diskname:size[:reserve]`

This keyword allows segments to reserve space on multiple disk partitions. The installation software will not split a segment across disk partitions, but the segment may do so in a `PostInstall` script. Use of multiple disk partitions is discouraged.

size and *reserve* have the same meanings as for `$DISK`. For UNIX platforms, *diskname* is either an explicit partition name (e.g., `/home2`) or an environment variable name of the form `DISK1`, `DISK2`, ... `DISK99`. The installation software will set the environment variables `DISK1`, `DISK2`, etc. to the absolute pathname for where space has been allocated. These environment variables are defined for `PreInstall` and `PostInstall`, but not for `DEINSTALL`. `$PARTITION` keywords are assumed to be in sequential order, so that environment variable `DISK1` will refer to the first keyword encountered, `DISK2` to the second, etc.

For NT platforms, *diskname* must be a disk drive name. For example,

```
$PARTITION:"D:":2048
```

requests 2MB of space on the "D" disk drive.

For example, suppose a Tactical Decision Aid (TDA) is compiled to run on an HP, a Solaris, and an NT platform. Assume for the HP it requires 512 K of memory, requires 1 Megabyte (MB) of disk storage for the program and its data files, and will expand over time to a maximum of 4 MB. For Solaris, assume it requires 576 K of memory, 1.5 MB for initial disk space, and will expand to 5 MB. For a PC, assume the requirements are the same as for the Solaris machine. The proper `Hardware` file is

```
#ifdef HP
   $CPU:HP
   $DISK:1024:3072
   $MEMORY:512
#elif SOL
   $CPU:SOL
   $DISK:1536:3584
   $MEMORY:500
#elif PC && NT
   $CPU:PC486
   $DISK:1536:3584
   $OPSYS:NT
   $MEMORY:571
#endif
```

Note that this example indicates that the information described is the same for all HP platforms, the same for all Solaris platforms, but that it only applies to PC486 or better machines running Windows NT.

As another example, assume a data segment is to be allocated across three disk partitions. Further assume that the first partition must be `/home5` and requires 10 MB, but the remaining space required is 20 MB each and can be on any available disk partition. The proper `$PARTITION` statements are:

```
$PARTITION:/home5:10240
$PARTITION:DISK2:20480
$PARTITION:DISK3:20480
```

Assume that the installation software is able to allocate space on `/home5` as requested, and that the remainder of the space requested is on `/home18`. The installation software will set the following environment variables:

```
setenv DISK1         /home5
setenv DISK2         /home18
setenv DISK3         /home18
```

`$TEMPSPACE:size`

Some segments may need temporary space during the installation process. The `$TEMPSPACE` keyword requests that *size* kilobytes of disk space be allocated for temporary use during the installation process. If space is available, the installation software sets the environment variable `COE_TMPSPACE` to the absolute path where space was allocated. If space is not available, an error message is displayed to the operator and the segment installation fails. The installation software automatically deletes the allocated space when segment installation is completed. Space is allocated prior to executing `PreInstall`.

5.5.2.15 Help

This segment descriptor is a place holder for a future COE revision. Its purpose is to provide a mechanism for identifying and managing help files within the system. Segment developers should use this descriptor now to reduce migration problems later.

As Figure 5-2 indicates, segment help files are located directly underneath the directory

`SegDir``/data/Help`

They are listed individually in the `Help` segment descriptor and grouped according to their format. Help file format is identified by one of the following keywords:

$HTML a list of help files in HTML format.

$MAN a list of help files in UNIX man page format.

$MSHELP a list of help files in Microsoft Help format (NT only).

$TEXT a list of help files in plain ASCII text format (i.e., no graphics or special characters).

$OTHER a list of files in a format other than that identified by the preceding keywords.

The order in which these keywords is listed is not important and they may be repeated multiple times within the segment descriptor. HTML is the COE-standard format, but the other formats are provided to assist legacy system migration.

For example, assume a segment contains two HTML-format help files (`H1` and `H2`), UNIX man pages (`man1` and `man2`), three ASCII text files (`T1`, `T2`, and `T3`), and one help file in an internal format (`doc1`). Then the proper `Help` segment descriptor entries are:

```
[Help]
$HTML
H1
H2
$MAN
man1
man2
$TEXT
T1
T2
T3
$OTHER
doc1
```

5.5.2.16 Icons

The `Icons` descriptor is used to define the icons that are to be made available on the desktop to launch segment functions. The format of the descriptor is a list of files underneath `data/Icons` that define icon bitmaps and their associated executables. Refer to the Executive Manager API documentation for a description of the file format.

5.5.2.17 Menus

Use the `Menus` descriptor to list the names of menu files contained within the segment. Figure 5-2 shows that segment menu files are located underneath `data/Menus`. Refer to the Executive Manager API documentation for the menu file format.

For account groups, this descriptor is simply a list of the account group's menu files. For other segments, the format of each line is

```
menu file[:affected menu file]
```

where *menu file* is the name of a menu file underneath the segment's `data/Menus` subdirectory, and *affected menu file* is the account group menu file to update. If multiple account groups are affected, as listed in the `SegName` descriptor, each affected account group is updated. If no affected menu file is listed,

then menu file is simply added to the list of menu files which comprise the account group's menu templates.

For example, suppose a segment called ASWTDA has four menu files in the data/Menus subdirectory: System, MoreStuff, ASWTDA, and Logging. Assume that System is to be added to the affected account group's System menu file, and MoreStuff is to be added to the affected account group's Default menu file. The proper entries are as follows:

```
System:System
MoreStuff:Default
ASWTDA
Logging
```

5.5.2.18 Network

The Network descriptor is used to describe network-related parameters. Use of this descriptor requires prior approval by the DII COE Chief Engineer and its use is restricted to COE-component segments, except for DCE Servers which are not necessarily COE-component segments. VerifySeg will strictly fail any segment that includes this descriptor unless it is a COE-component segment or it is a DCE server.

One or more entries may follow each keyword listed below.

$HOSTS

IP addresses and hostnames are generally established by a system or network administrator. Segments may add IP addresses and host names as follows:

```
$HOSTS
LOCAL | REMOTE :IP address:name{:alias}
```

where *IP address*, *name*, and *alias* are as defined for the UNIX /etc/hosts file. If the IP address specified already exists in the network hosts file, the name and alias entries are added as alias names. If LOCAL is specified, the entry is made only to the local network hosts file. If REMOTE is specified, the entry is applied to the NIS/NIS+ or Domain Name Service (DNS) server. If REMOTE is specified but neither NIS/NIS+ or DNS are installed, it will default to LOCAL.

Segments should rarely need to directly add host table entries. VerifySeg will issue a warning for any segment which adds host table entries.

$KEY:Network:key

key is the authorization key given to the segment developer by the Chief Engineer. This entry is required only once within the section, and it applies to all entries within the section.

$MOUNT (UNIX only)

The $MOUNT keyword is used to specify NFS mount points. The syntax is

```
hostname:NFS mount point:target dir
```

where *hostname* is the name of a platform on the network, *NFS mount point* is the file partition to mount, and *target dir* is where to mount the requested partition on the local machine. If target dir does not exist on the local machine, it will be created.

For example, the sequence

```
$MOUNT
dbserver:/home3/USERS:/h/USERS
```

will perform the UNIX equivalent of

```
mount dbserver:/home3/USERS /h/USERS
```

If the hostname specified is the same as the local machine, a mount is not performed. Instead, the NFS mount point is made available for other platforms to mount. If a mount is performed as a result of processing this keyword, the system will automatically reboot the system after segment installation is completed. It performs as if the `$REBOOT` keyword (see the `Direct` descriptor) were encountered; that is, the operator is notified that a reboot is required and given an option to override the reboot directive.

`$NETMASK:mask`

This keyword allows a COE-component segment to set the subnet mask to *mask*. This should rarely be required since the netmask is normally established as part of the COE kernel. If two COE-component segments attempt to set the netmask, the last segment loaded succeeds.

`$SERVERS`

Most COE services are implemented as servers. This keyword allows a segment to list the non-DCE servers, by symbolic name, that it contains. These servers are registered with the COE so that other segments can obtain their location through the `COEFindServer` function (see Appendix C).

> **Note:** Servers implemented through DCE functions should not use this keyword. The `DCEServerDef` descriptor should be used instead.

Each name listed is added to a table maintained by the COE of all servers in the system. This table is used by the System Administration software to allow a site administrator to indicate which platform actually contains the server. The name given is added as an alias to the network host table for the platform that contains the server. If NIS/NIS+/DNS are being used, the alias is added to the NIS/NIS+/DNS-managed host table. Otherwise, it is added to `/etc/hosts`.

For example, assume a COE-component segment contains two servers named `masterTrk` and `masterComms`. Assume that this segment is loaded on two workstations: `sys1` and `garland`. Some servers are designed to recognize whether they are the master server or are a slave to a master server located elsewhere. For this reason, the COE must handle situations where the same segment is loaded on a server and a client machine. Assume in this example that the segment operates as a master server on `sys1`, but as a slave on `garland`.

The following statements identify the servers contained within this segment:

```
$SERVERS
masterTrk
masterComms
```

When the segment is loaded, the installation software performs the following actions:

1. Add `masterTrk` and `masterComms` to the COE-maintained list of servers if they are not already there.
2. Check to see if `masterTrk` or `masterComms` already exist in the network host table. If so, processing is completed.

3. Otherwise, ask the operator if this is the server platform for `masterTrk` and `masterComms`.
4. If the operator answers "no" to the previous question, processing is complete.
5. If the answer is "yes," update the network host table to contain `masterTrk` and `masterComms` as aliases for the machine being loaded.

Note that this approach does not require the server (`sys1`) to be loaded prior to the client (`garland`). Furthermore, the site administrator can later change the configuration because all necessary information is available to the System Administrator software. Also note that the segment does not require the actual hostnames or IP addresses.

Hostnames are site-specific and cannot be predicted in advance. Therefore, the COE requires that segments use meaningful symbolic names as illustrated here instead of making assumptions about a specific hostname or naming convention.

5.5.2.19 Permissions

The Security Administrator software provides the ability to describe objects (files, data fields, executables, etc.) which are to be protected from general access. This information is used to create profiles which limit an operator's ability to read or modify files. Applications may query the security software to determine the access permissions granted to the current user. The `Permissions` file is the mechanism by which segments describe objects and what permissions to grant or deny for the objects.

This descriptor is a sequence of lines of the form:

```
object name:permission abbreviation:permission
```

object name is the item to be controlled, *permission* is the type of access to grant or deny (add, delete, read, etc.), and *permission abbreviation* is a single character abbreviation for the permission.

Permission abbreviations specified for an account group must agree with all segments which become part of the group. The following are reserved abbreviations and their meanings:

A - Add
D - Delete
E - Edit
P - Print
R - Read
V - View
X - Transmit

Segments may use additional abbreviations as required.

For example, suppose a segment generates reports that are to be protected. Permissions relevant to reports are delete, print, read, and archive. The proper `Permissions` file is:

```
Reports:D:Delete:P:Print:R:Read:Z:Archive
```

(Z is used to indicate archive permission in this example.)

If the `Permissions` file is missing, the security software will report that no access permissions are to be granted for the requested object.

5.5.2.20 Processes

Use the `Processes` descriptor to identify non-DCE background processes (see subsection 5.10.6). The format of the descriptor is a keyword which identifies the type of process, followed by a list of processes to launch in the form

```
process {parameters}
```

where *process* is the name of the executable to launch, and *parameters* are optional process-dependent parameters. Output from the process is piped to `/dev/null`. For example, suppose `TestProc` is a background process which accepts two parameters, `-t` and `-c`. It will be launched in a manner equivalent to

```
TestProc -t -c >& /dev/null &
```

Valid keywords to identify process type are:

$BOOT	specify a list of processes to launch at boot time
$BACKGROUND	specify a list of background processes
$PERIODIC	specify a list of background processes to run at some specified interval
$PRIVILEGED	specify a list of processes to run in privileged (i.e., "root") mode (available for UNIX only)
$RUN_ONCE	specify a list of "one-shot" processes to run the next time the system is started, but only the next time the system is started and never thereafter
$SESSION	specify a list of login session processes
$SESSION_EXIT	specify a list of processes to run prior to terminating a login session

The `$PERIODIC` keyword requires specification of the required interval, in hours. The format is

```
$PERIODIC:hours
```

where *hours* is a decimal value.

Executables are assumed to be in the segment's `bin` subdirectory. The `$PATH` keyword can be used to indicate a different location. The syntax for the `$PATH` keyword is

```
$PATH:pathname
```

where *pathname* may be either a relative or an absolute pathname. If the pathname is relative, it is relative to the segment's home directory.

Use of boot-time, background, periodic, privileged, and "one shot" processes requires authorization by the Chief Engineer. Therefore, the `$KEY` keyword must be specified once, in the form

```
$KEY:Processes:key
```

The authorization key applies to all requests within the `Processes` segment descriptor.

The `Processes` descriptor is a powerful capability the COE provides for managing application processes. Refer to documentation in the Developer's Toolkit for more detailed information on this descriptor.

> **Note:** DCE processes are not described with the `Processes` descriptor. Use the applicable DCE keywords within `DCEServerDef` and `DCEClientDef` instead.

5.5.2.21 Registry (NT only)

The `Registry` segment descriptor allows segments to add entries to the NT registry. It is followed by a list of keys and filenames, underneath the segment's `data/Registry` subdirectory, whose contents are the key values to add to the registry. `VerifySeg` will generate an error if any of the files listed do not exist.

The parameters for this keyword are

```
keyloc:registry description file
```

where *keyloc* is the root location in the registry to add key values found in the file *registry description file*. At present, *keyloc* may have only the value

`$HKEY_LOCAL_MACHINE\SOFTWARE\COE`.

Future revisions may expand the *keyloc* parameter.

Consider the following example.

```
[Registry]
$HKEY_LOCAL_MACHINE\SOFTWARE\COE:MyEntries
```

This indicates that the segment contains a file named `MyEntries` located under the directory *`SegDir`*`/data/Registry` (where *SegDir* is the segment's assigned directory). The contents of the file `MyEntries` will be added to the registry under the key

`HKEY_LOCAL_MACHINE\SOFTWARE\COE\`*`SegType`*`\`*`SegDir`*

where *SegType* is the segment's type and *SegDir* is the segment's assigned directory.

Following is the format of the registry description file:

```
$KEY:key-name
$STRING:Name:StringValue|$BINARY:Name:BinaryValue|$DWORD:Name:DwordValue
```

where *key-name* is the name of the subkey to create beneath

`keyloc``\`*`SegType`*`\`*`SegDir`*

- *key-names* may include '\'s to indicate that subkeys are to be created.
- The `$STRING`, `$BINARY`, and `$DWORD` keywords signify a string, binary or double-word name/value pair that is to be maintained beneath the given key. The given *Name* follows the keyword and then the *value* follows.
- At least one `$KEY` must be specified in the registry description file. Multiple `$KEY`'s may be specified in the registry description.
- All `$STRING`, `$BINARY`, and `$DWORD` settings must appear at the beginning of a line. These settings are not required and if omitted the given key will be created without any name/value pairs. There may be multiple `$STRING`, `$BINARY`, and `$DWORD` settings per `$KEY` and the order in which they are listed is not important.

The following example is for a software segment whose segment directory is `SegA`. Assume that key values are in the file `settings.dat` located underneath the directory `SegA/data/Registry`. The proper `Registry` descriptor entry is

```
[Registry]
$HKEY_LOCAL_MACHINE\SOFTWARE\COE:settings.dat
```

The following are example entries for `settings.dat`:

```
$KEY:Analyze
$STRING:ControlFile:\Program\Analyze\Control.dat
$DWORD:UsageCount:0
$KEY:Defragment
$STRING:ControlFile:\Program\Defragment\Control.dat
$DWORD:UsageCount:0
$KEY:Reporting
$STRING:ControlFile:\Program\Report\Control.dat
$STRING:Example1:Callsign is Foxtrot Tango 3
$STRING:Example2:Response is “Spring time 3!”
$DWORD:UsageCount:21
$BINARY:Encoding:17
# Here are several keys with no name/value pairs that also
# illustrates creating subkeys
$KEY:Reporting\Type1
$KEY:Reporting\Type2
$KEY:Reporting\Type3
```

The above example creates the following registry entries:

```
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Analyze
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Defragment
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Reporting
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Reporting\Type1
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Reporting\Type2
\HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA\Reporting\Type3
```

Note that the values given for both the `$DWORD` and `$BINARY` parameters are given in decimal format, but will appear in hexadecimal format (`$DWORD`) and binary format (`$BINARY`) when viewed from the NT registry editor window.

The registry capability must be used with great care.

- The installer tools will remove registry entries added with this segment descriptor when the segment is deleted.

- Segment developers shall not create root keys.

5.5.2.22 ReqrdScripts (UNIX only)

Use the `ReqrdScripts` descriptor to define the files that establish the runtime environment (account group segment types) or to define files to extend the runtime environment (all other segment types). For account group segments, the syntax is one or more lines of the form:

```
script name:C | L
```

where *C* means to copy and *L* means to create a symbolic link. This flag is used when login accounts are created to either copy environment files to the user's login directory or to create a symbolic link. There can be a maximum of 32 scripts. A script name is restricted to a maximum length of 32 characters.

For example, the `ReqrdScripts` file for the System Administrator account group is

```
.cshrc:C
.login:C
```

The descriptor format for segment types other than account group is slightly different:

```
script name:env ext name
```

where *script name* is the name of a script in the affected account group's `Scripts` subdirectory and *env ext name* is the name of an environment extension file in the present segment's `Scripts` subdirectory.

For example, assume a segment loaded under `/h/TstSeg` with a segment prefix `TST` is to be added to the System Administrator application and it requires extending the `.cshrc` file. The proper `ReqrdScripts` entry is:

```
.cshrc:.cshrc.TST
```

The installation tools will insert the statements

```
if (-e /h/TstSeg/Scripts/.cshrc.TST) then
   source /h/TstSeg/Scripts/.cshrc.TST
endif
```

into the file `/h/AcctGrps/SysAdm/Scripts/.cshrc`. When the segment `TstSeg` is deleted, the installation tools will remove these statements.

Refer to documentation in the Developer's Toolkit for more information.

5.5.2.23 Requires

Segment dependencies are stated through the `Requires` descriptor. The format is:

```
[$HOME_DIR:pathname]
[$LIB:library name[:library path]]
segment name:prefix:home dir:[version{:patch}]
```

Segments will not be loaded until all segments they depend upon are loaded. For this reason, the parent segment for an aggregate must *not* list child segments in the `Requires` descriptor.

> **Note:** The parent segment for a child does not need to be listed in the child's `Requires` descriptor. By virtue of naming the aggregate parent in `SegName`, there is an implied dependency.

The optional `$HOME_DIR` keyword is used in situations where a segment must be loaded onto the disk in a particular place. This technique should be avoided.

The optional `$LIB` keyword is used to identify a dependency on shared libraries. *library name* describes the shared library or Dynamic Link Library (DLL) on which the segment is dependent. The shared file is

normally located in the dependent segment's `bin` directory; however, *library path* can be used to define a different path for the shared file.

For example, assume the segment `TEST` must be installed in the directory `/home3/tmp/TEST`, it requires version 3.0.2 of segment `SegA` with patches P1 and P4, and also requires `SegB` version 5.1.1. The `Requires` descriptor is

```
$HOME_DIR:/home3/tmp/TEST
SegA Name:SEGA:/h/SegA:3.0.2:P1:P4
SegB Name:SEGB:/h/SegB:5.1.1
```

In some cases, it may be possible that a segment dependency can be fulfilled by one or more segments. This is indicated by bracketing such segments with braces and using the keyword `$OR` between acceptable alternatives.

As an example, suppose the segment `TEST` above has a dependency that can be satisfied by `SegA` or the combination of `SegB` and `SegC`. The proper `Requires` descriptor is

```
$HOME_DIR:/home3/tmp/TEST
{
   SegA Name:SEGA:/h/SegA
$OR
   SegB Name:SEGB:/h/SegB
   SegC Name:SEGC:/h/SegC
}
```

Multiple bracketed alternatives may appear in the same descriptor.

5.5.2.24 Security

The `Security` descriptor is of the following form

```
classification{:caveat}
```

where *classification* indicates the highest classification level for the segment (`UNCLASS`, `CONFIDENTIAL`, `SECRET`, `TOP SECRET`). The optional list of *caveats* is used to document releasability restrictions. If the segment contains items with multiple classification levels, the highest classification level must be specified. If the segment has multiple releasability restrictions, the most restrictive ones should be listed as caveats.

> **Note:** This descriptor is required and its purpose is primarily for documentation. Caveats are not used for any other purpose but the classification is used by the installation tools to determine whether or not a segment should be loaded onto a platform. The segment's classification level is compared against the platform's current classification level (as displayed in the security banner) and is not loaded unless the platform level dominates the segment classification level. This feature is *not* to be considered a trusted capability but is merely provided as an aid to the installer. The classification and caveat must *not* be confused with data labeling or other security features provided by trusted systems.

5.5.2.25 SharedFile

This segment descriptor handles installation of NT shared DLLs and UNIX shared libraries. It is followed by a list of filenames that are the names of the shared libraries (UNIX) or DLLs. They must be located in the segment's `bin` subdirectory, which is the DII-compliant location for shared files. `VerifySeg` issues an error message if a filename listed does not exist under the segment's `bin` subdirectory. Shared files must use the segment prefix naming convention to assure that the names are unique.

The `SharedFile` descriptor accepts two keywords: `$FILENAME` which is required and `$PATH` which is optional. The format for each follows.

`$FILENAME:filename`

This keyword establishes the shared library or DLL filename (parameter *filename*).

`$PATH:pathname`

This is an optional keyword which provides the directory path *pathname* of the file when it is not located in the segment's `bin` directory.

> **Note:** The path is *very important* in a UNIX environment as the shared library must be placed in the same location as when the executable binary was created; otherwise, the binary will not execute.

At installation time, the segment installer copies the shared file to the directory `/h/COE/Shared`, deletes the shared file from the segment's `bin` subdirectory, and then creates a symbolic link from `/h/COE/Shared` to the original location. This is done so that the search path for finding shared files does not need to include any entry other than `/h/COE/Shared`. Segments which have a dependency upon the shared file must identify the segment which provides the shared file in the `Requires` segment descriptor.

Installation requires special care to ensure that a segment which provides a shared library/DLL is not removed when there are segments still installed that require it. For this reason, the installer maintains a usage counter for the shared file. When the segment which "owns" it is installed, the count is set to 1. As segments which depend upon it are installed or removed, the counter is incremented or decremented as appropriate. The installation tools thus prevent the "owning" segment from being removed until the usage count indicates there are no more dependent segments installed.

Shared libraries/DLLs require specific consideration within the COE.

- Segments must state dependencies on the segment providing the shared library/DLL, not the actual file itself.
- One segment may not update a shared library/DLL "owned" by another segment. This would otherwise contradict the fundament COE principle that objects (resources, files, etc.) may be modified only by the segment which owns the object, or by the COE.

5.6 Segment Installation

Segment installation requires some form of electronic media (tape, CDROM, disk, etc.) that contains the segments, and that has a table of contents which lists the available segments. `MakeInstall` is the tool which creates such electronic media. However, it is important to identify the operations (e.g., compression) performed on segments and the sequence in which these operations are performed.

Installation requires reading the table of contents created by `MakeInstall`, selecting the segments or Configuration Definitions to install, and then copying the segments to disk. Segments may actively participate in the installation process through `PostInstall`, `PreInstall`, and `DEINSTALL` scripts. This subsection details both the `MakeInstall` tool and the installation sequence. At the end of this subsection, detailed information on database creation and deinstallation is presented.

5.6.1 MakeInstall Flowchart

Figure 5-14 shows the sequence of operations performed by the `MakeInstall` tool.

1. `MakeInstall` is given a list of segments that are to be processed. For each segment in the list:
   a) If the segment is not already on disk, it is extracted from the repository and placed in a temporary location.
   b) A check is made to ensure that the segment is a valid segment.
   c) If the segment is invalid, an error message is displayed. If the segment was checked out of the repository and placed in a temporary location, the temporary segment is deleted. `MakeInstall` then terminates.

2. If all segments are valid, a worklist is created. The worklist is sorted to ensure that segments which have dependencies appear in the list *after* the segments they depend upon. This ensures that at install time a tape will not have to be rewound because of segment dependencies. Note that specification of an aggregate automatically includes each child. The order in which child segments are placed onto the distribution media is *not* guaranteed but is normally the order in which they are specified by the parent segment.

3. For all segments in the worklist:
   a) Prepare the segment by executing the segment's `PreMakeInst` descriptor if it exists. `PreMakeInst` is prevented from modifying the segment's `SegDescrip`. Otherwise, `PreMakeInst` could invalidate the segment validation step above.
   b) Unless the segment specifies otherwise, all segment subdirectories except `SegDescrip` are compressed.
   c) The compressed segment and its descriptor directory are written out to the specified electronic media.
   d) If the segment was extracted from the repository and placed in a temporary location, the temporary segment is deleted.

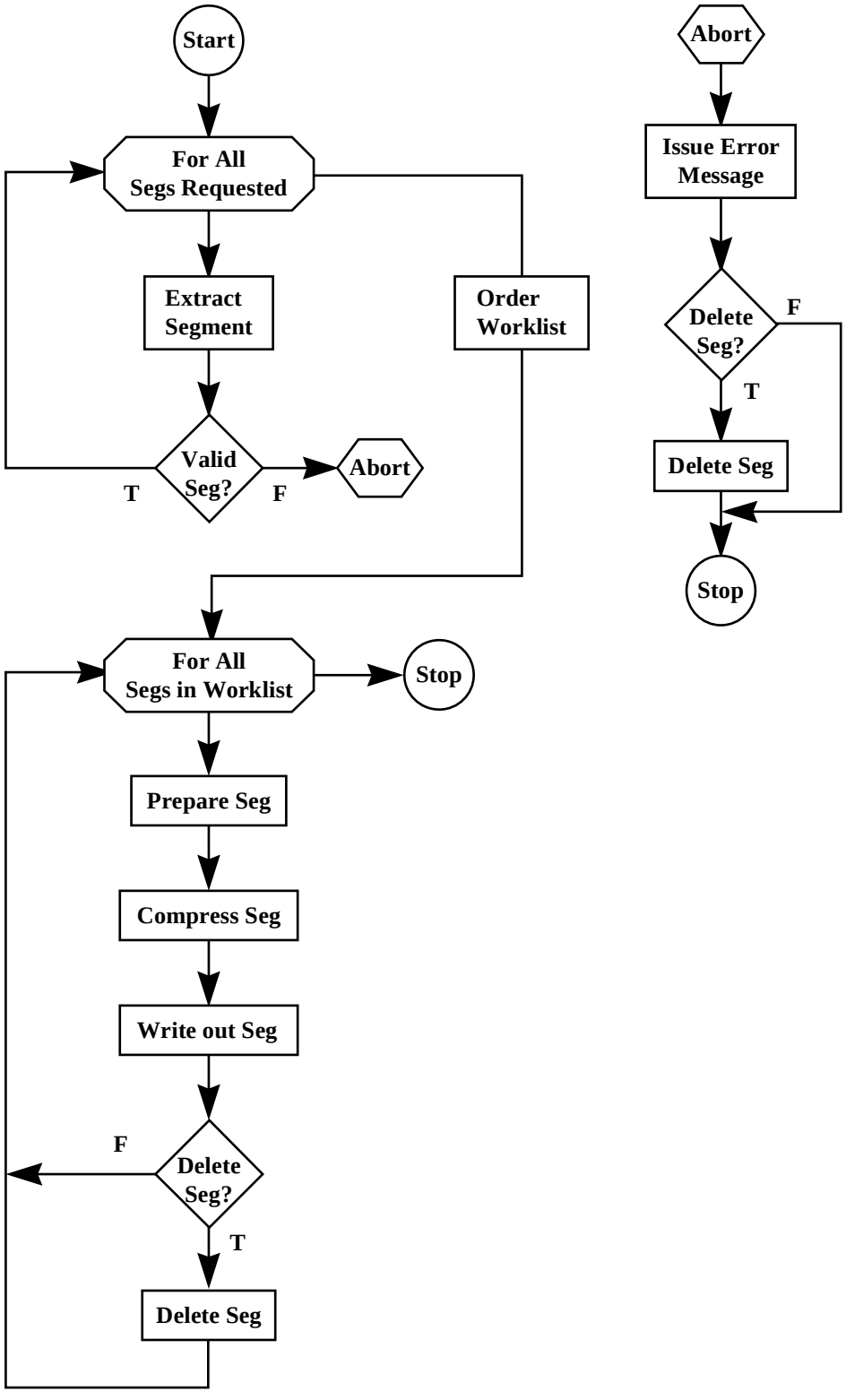


Figure 5-14: MakeInstall Flowchart

5.6.2 Installation Flowchart

Figure 5-15 is a detailed flowchart for the segment installation process. The sequence of `PreInstall`, `PostInstall`, and `DEINSTALL` executions is the most significant aspect of the flowchart. Directives contained in the `Direct` descriptor may affect the sequence (e.g., use of `$REBOOT` and `$ROOT` keywords), but such details are omitted for clarity. The installation software automatically removes patches when a segment is replaced and deletes any temporary space (`$TEMPSPACE` keyword) allocated for the segment. These details are also omitted for clarity.

1. A load device is selected (tape, disk, etc.) and the table of contents created by `MakeInstall` is read.

2. Segments found in the table of contents which do not match the target platform are removed from consideration. Similarly, a check is made to ensure that an operator cannot inadvertently load a segment for which he is not authorized. The environment variables `MACHINE_CPU` and `MACHINE_OS` are set to indicate the hardware platform.

3. The media may have Configuration Definitions defined. If they are defined:

   a) The operator may select a Configuration Definition to load.
   b) If a custom installation is desired, the operator is presented with the table of contents in which all segments in the selected Configuration Definition are highlighted. The operator may add or delete segments from this list.
   c) If Configuration Definitions are not defined, the operator is shown the table of contents and must manually select the desired segments.

4. For all segments selected, a check is made to see if the segment is loadable. To be loadable, all dependent segments must either be selected or already on disk. Conflicting segments must not be selected, nor may they already have been loaded on disk.

5. For all segments selected:

   a) The installation tools determine where to load the segment. The environment variable `INSTALL_DIR` is set to the absolute pathname to where the segment will be loaded. Segments can *not* assume that any environment variables other than `MACHINE_CPU`, `MACHINE_OS`, `SYSTEM_ROOT` (for NT only), `INSTALL_DIR`, and those set to refer to disk space (`COE_TMPSPACE`, `DISK1`, etc.) are defined.
   b) If an old version of the segment already exists on disk, the old segment's `DEINSTALL` script is run.
   c) The new segment's `PreInstall` script is loaded and executed. Note that the new segment is *not* yet on disk.
   d) The old segment is deinstalled by the installation tools. Modifications made through the descriptor files are reversed.
   e) The old segment is deleted from disk.
   f) The new segment is loaded from tape onto disk and decompressed if necessary.
   g) The installation tools process commands from the new segment's descriptor files.
   h) The new segment's `PostInstall` script is run. `PostInstall` may invoke runtime tools described in Appendix C (e.g., to prompt the user).
   i) A status message is displayed indicating whether or not the segment was successfully installed.

6. If any of the segments installed requested a reboot, the operator is notified and asked for confirmation. If the operator confirms, the system is rebooted.

5.6.3 Database Installation and Removal

Within the overall installation and removal flowchart presented in Figure 5-15, there are some special considerations with regards to handling SHADE databases. Database installation is described first, then database deinstallation.

5.6.3.1 Database Installation

This subsection describes the installation process flow and how the database segment components work together to install a data store on the COE database server. `PostInstall`, automatically invoked by `COEInstaller`, drives the actual installation and creation of the database and its storage by executing the scripts residing under the install directory of a database segment. The flowchart in Figure 5-16 depicts the process logic of a `PostInstall` file with regards to database segments.

The DBMS should be operating in its maintenance mode (e.g. Oracle's command `STARTUP DBA EXCLUSIVE`) when a database segment or database patch segment is installed. This prevents users from accessing data objects during their creation and possibly corrupting either the segment or the database instance.

Table 5-8 shows, in broad outline, the sequence of steps performed by a database server segment when it is creating the database. It uses Oracle and Sybase as examples. The first three steps must be performed by a database account with DBA privileges. The owner account (and there may be more than one) should be restricted so it can only create objects in the data stores designated for its use. The remaining steps should be performed by the owning account and should be done without DBA privileges. This ensures that data objects are not inadvertently created in data stores belonging to other databases.

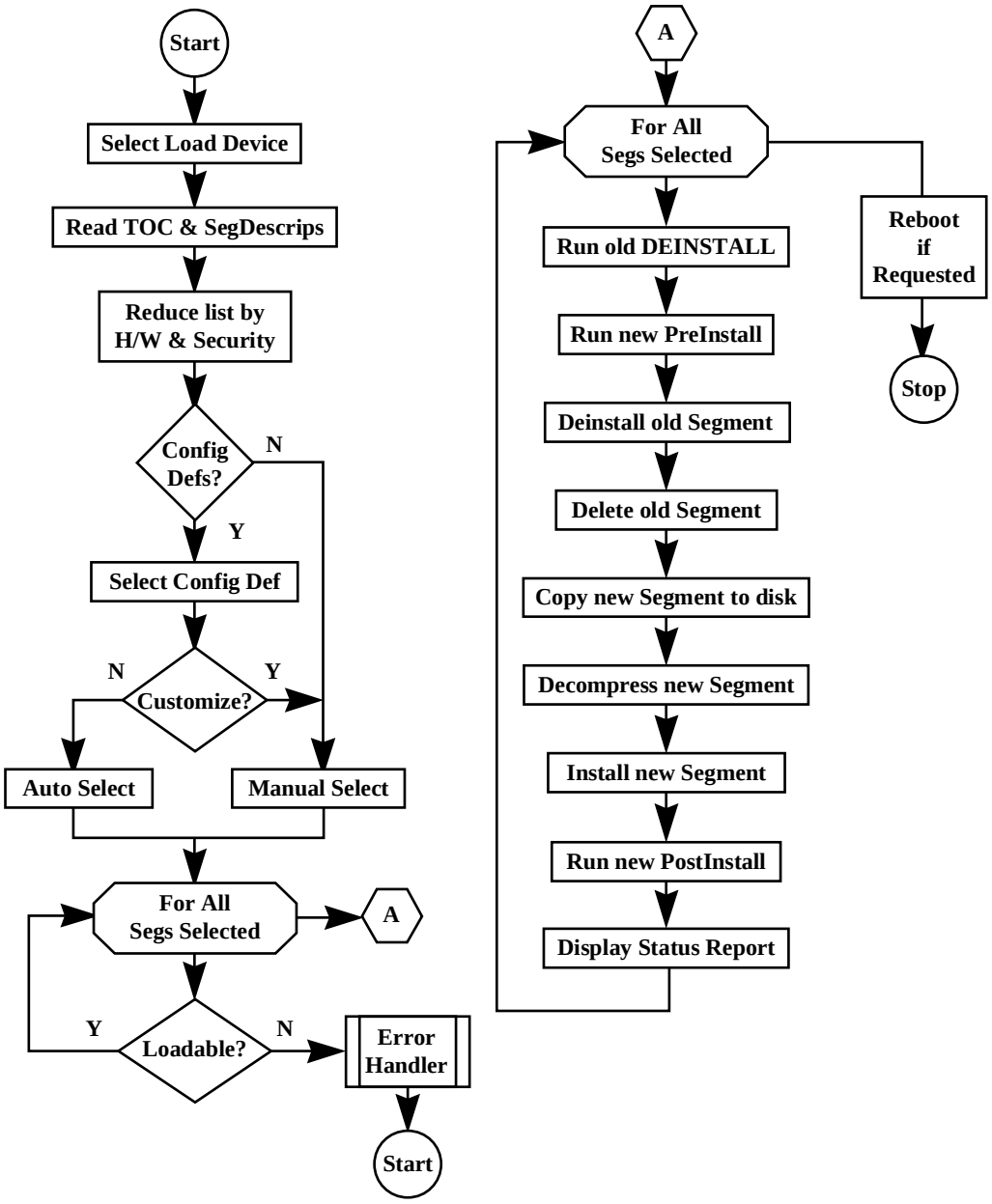


Figure 5-15: Installation Flowchart

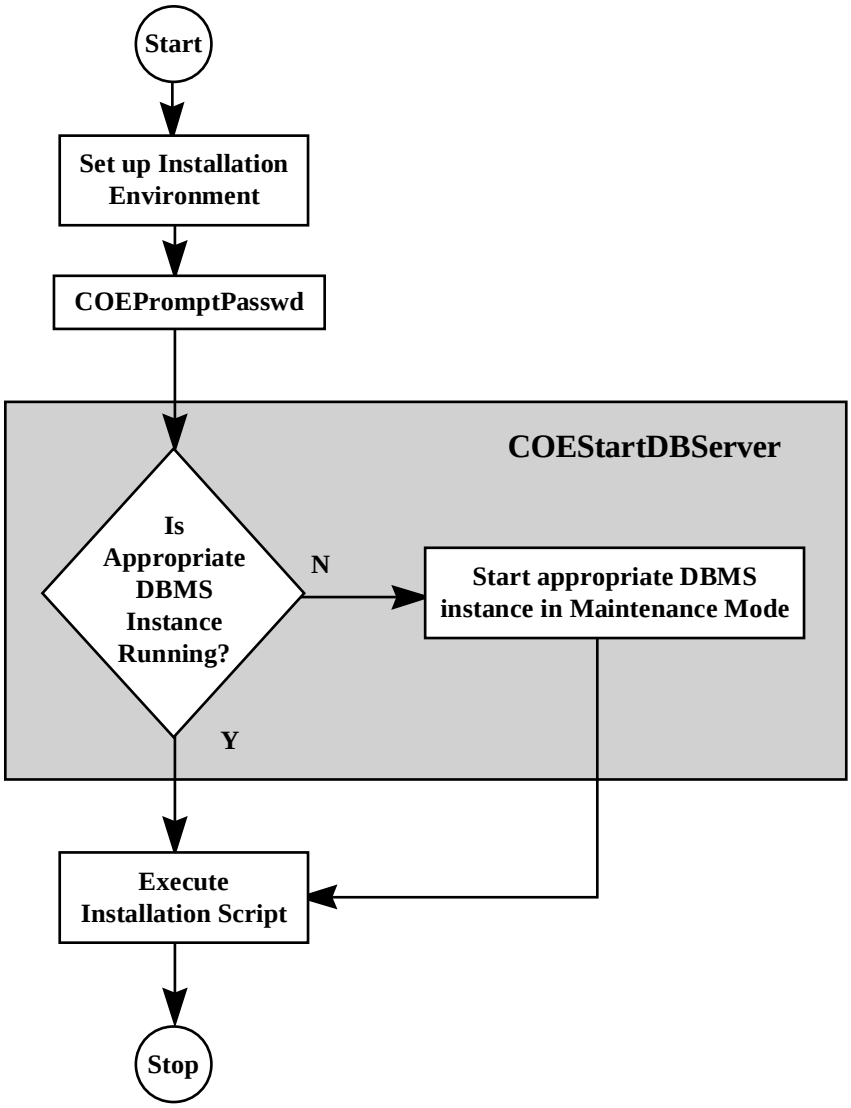


Figure 5-16: PostInstall Logic for DB Install

Function	User	Oracle SQL Command	Sybase SQL Command
1. Allocate Storage	DBA	create tablespace ... datafile ...	create database…
2. Create Owner	DBA	create user ...	
3. Create Role(s)	DBA	create role ...	create group …
4. Create Database	Owner	create schema	create table …
5. Load Data	Owner	insert into table	insert into table
6. Create Constraints	Owner	alter table … add constraint	create constraint …
7. Grant Access	Owner	grant ... on table ... to role	grant ... on table ... to group
8. Disconnect Owner	DBA	revoke CONNECT from ...	

Table 5-8: Application Database Creation

1. **Allocate Storage**. This step is performed by the DBA and creates the physical storage needed for the database. Developers shall not assume any particular disk configuration when creating data files and shall create all files in the segment's `DBS_files` subdirectory. Developers may create multiple storage areas (e.g., Oracle tablespaces or Sybase segments) to separate different groups of data objects. Developers shall not modify the core database storage areas.

2. **Create Database Owner**. This step is performed by the DBA and creates the account or accounts that will own the data objects. Their access will be limited to the storage areas created by the segment and to public storage areas (e.g. Oracle tablespace `TEMP` or `USERS`). Owners shall not have access to system storage areas (e.g. Oracle tablespace `SYSTEM`). No permanent objects shall be created in public storage areas by database segments. No objects shall be created in system storage areas. Owners shall not have database administrator privileges.

3. **Create Database Roles**. This step is performed by the DBA and creates the database roles necessary to manage user access. Developers should match the role definitions to the access needed by applications. Developers should not grant privileges that allow users to manipulate the data objects' structure (e.g. Oracle's `Alter` privilege). Users should not be allowed to create their own indexes either.

4. **Create Database**. This step is performed by the Owner and creates tables, views, indexes, constraints, sequences, and any other data objects that are part of the database. If the developer has defined multiple owners, a separate script should be provided for each one. No objects will be created that will be owned by the DBMS default accounts (Oracle's `SYS` or `SYSTEM`, Sybase's `sa`) or by any other account intended to be a DBA. Creation of constraints and indexes may be deferred to speed the data load.

5. **Load Data**. This step is performed by the Owner and fills the data objects previously created. Although index and constraint creation were defined as occurring in the previous step, developers may defer them until the data load is complete to improve performance.

6. **Create Constraints**. This step is performed by the Owner and creates any indexes, constraints, triggers, or other objects that are part of the database but whose creation was deferred until after the data load.

7. **Assign Grants**. This step is performed by the Owner and grants the appropriate access permissions on data objects to the database roles previously defined. Grants shall not be made directly to users accounts. Grants shall not be made to general purpose users (e.g. Oracle's `PUBLIC` user). Only the

owner or the DBA are allowed to administer grants. Other users will not be given permissions to further disseminate grants.

8. **Disconnect Owner**. The last step – revoking database connection privileges from the owner upon completion of the load process – is performed by the DBA. It ensures that users cannot connect to the database as the owner of the data and thereby prevents users from modifying schemas, indexes, or grants. Developers shall also require the database administrators to change the password of the owner account upon completion of the database creation.

The flowchart in Figure 5-17 depicts the processing logic of the `install` directory's scripts which drive the creation of the database objects. Each package `install` script executes the database definition scripts that connect to the COE Database Server to create database objects and perform other data definition functions.

The package `install` script executes database definition scripts that actually connect to the COE DBMS Server to create the database objects and perform other data definition functions.

5.6.3.2 Database Segment Deinstall

Deinstallation has a different flavor with databases. First, databases are dynamic. As users make changes to their databases, sites' data sets will diverge from each other. It is unlikely that any two operational sites will have exactly the same data at any point in time. Second, inter-database dependencies restrict the ability to remove segments in a modular way.

However, developers need to provide the capability to remove the application's server segment from the Database Server. This means removing the database and all traces of its presence from within the DBMS and removing all files from the Database Server. The following steps, at a minimum, must be accomplished. Note that the remove storage step de-assigns the data files from the DBMS, it does not actually remove them from disk. The last step, remove files, is executed from the operating system to delete the data files. Table 5-9 illustrates the logic required, using Oracle as an example.

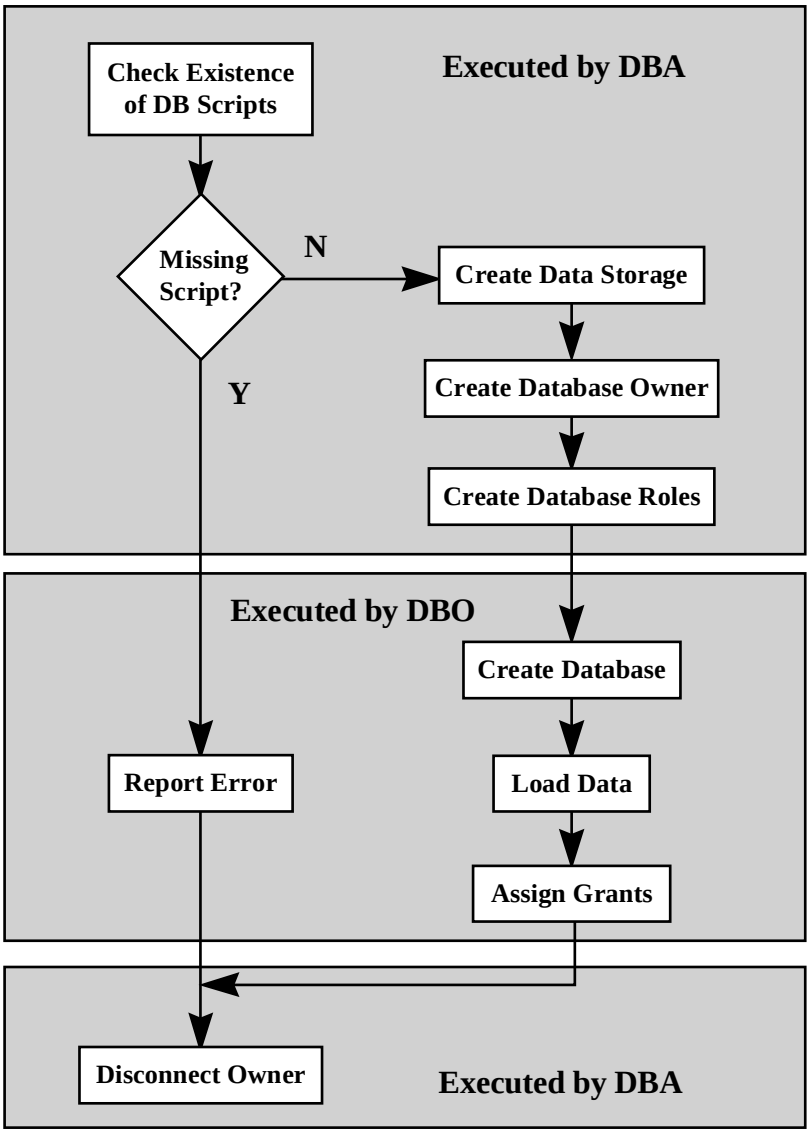


Figure 5-17: Install Scripts Logic

Function	User	Oracle SQL Command
Remove roles	DBA	drop role ...
Remove objects	owner	drop schema ...
Remove storage	DBA	drop tablespace ...
Remove owner	DBA	drop user ...
Remove files	DBA	N/A (Use OS commands)

Table 5-9: Application Database Deinstall

Within the Oracle server, combining the removal of storage and of data objects by using the Oracle command 'drop tablespace *x* including contents' is not recommended because it tends to overload the DBMS' rollback segments. Developers should use the 'drop schema' command followed by a 'drop tablespace' command instead.

When `DEINSTALL` is being executed to support a segment upgrade or patch, the upgrade or patch must support the deinstall/reinstall of data and supply the scripts to do so.

`DEINSTALL` scripts must be set up to fail nondestructively if other database segments are dependent on the segment to be deinstalled. This can usually be accomplished using the COE Tool `COELstDBDepends`.

5.7 Partial Segmentation for COTS Products

The segmentation process has several benefits, including the ability to state dependencies of one segment on another, which significantly simplify the installation process. From a macro perspective, the segmentation process is a matter of creating the appropriate segment descriptors to describe the segment and then running the `MakeInstall` tool to package the segment along with its segment descriptors. However, there are situations in which it is not convenient to physically repackage the application in order to put it into segment format. This is particularly true with large COTS products which are distributed on media such as CDROM or in a format provided by the vendor.

The DII COE provides a mechanism, called *partial segmentation*, which allows use of the COTS vendor's original distribution media and scheme while yet retaining the advantages of using segment descriptors to specify dependencies, identify conflicts, etc. In concept, the approach is to load a "pseudo-segment" which contains only the segment descriptors and use the vendor's installation process for the software itself. This allows the installation tools to verify that sufficient space exists, that dependencies are met, and that conflicts are resolved prior to loading the application.

Partial segmentation for COTS products requires that several actions be performed to ensure that it works properly.

1. A "pseudo-segment" must be created. This is done by creating a directory with the required segment descriptors which will give the segment a name, prefix, version number, etc. This must be registered as is any other segment. The version number for the "pseudo-segment" must include a primary version number that is used to track changes in the pseudo-segment and a secondary version number that is the COTS product's version number as provided by the vendor.

2. A `PreInstall` descriptor must be created which checks to see if a correct version of the COTS product is already installed. If it is not, the `PreInstall` must notify the user that the COTS product must be installed before continuing and then the `PreInstall` descriptor must return a failure status to the installer tool. This requires the operator to use the vendor supplied instructions to install the product before continuing.

3. Developers who use the partial segmentation process must certify in the *Version Description Document* delivered to the government that the installation will fail if the "wrong" version of the COTS product is installed. That is, if the pseudo-segment has been produced for version 3.2.1 of a COTS product but the user installs version 3.1.5 then this error must be detected by the `PreInstall` descriptor and handled accordingly.

4. Developers who use the partial segmentation process must provide a copy of the COTS product for testing and must make it clear how testers should process the copy to make it ready for installation.

The tool `COEScanCOTS` described in Appendix C is a slight variation on the partial segmentation process. It is specially designed for use in the NT environment where COTS products may have already been installed on the platform prior to the installation of the COE. This tool creates segment descriptor information for applications already installed and thus allows segments loaded subsequently to state dependencies on COTS products already installed.

> **Note:** Partial segmentation is supported but it is not normally the recommended approach for COTS products. Complete segmentation allows one to take full advantage of the benefits of the segmentation concept and process. Use of the partial segmentation approach requires prior approval by the cognizant DOD system engineer.

5.8 Security Considerations

COE-based systems typically operate in a classified environment. Therefore, the COE and the segment developer both must address security considerations. This section describes the security implications from a runtime environment perspective. It does not address procedural issues such as proper labeling of electronic media, requirements for maintaining paper trails showing originating authority, etc.

Certain restrictions described below are a result of how the operating system manages file versus directory permissions. The most specific permission (i.e., on a file) does not consistently override the least specific permission (i.e., on the file's parent directory).

This section is evolving as security policies are developed for COE-based systems and as legacy systems migrate to the COE. Further guidance will be issued as appropriate. Refer to the DII COE Chief Engineer for specific security concerns or for guidance in segment development beyond the information contained here.

5.8.1 Segment Packaging

Segments shall not mix classification levels within the same segment. It is permissible to create an aggregate that contains segments that are at different classification levels, but the parent segment must dominate the security level of any child segments.

Features that are not releasable to foreign nationals shall be clearly identified through documents submitted to the cognizant DOD SSA when the segment is delivered. Software and data that contain non-releasable features shall be constructed so that the features may be removed as separate segments.

All classified data shall be constructed as separate segments. Developers shall submit unclassified sample data to the SSA in a separate segment for the SSA to use during the testing process.

5.8.2 Classification Identification

All segments shall identify the segment's highest classification level in the Security descriptor. Developers shall submit documentation to the SSA that clearly identifies what features are classified and at what classification level.

5.8.3 Auditing

Segments that write audit information to the security audit log shall include the segment prefix in the output. This is required so that audit information can be traced to a specific segment.

5.8.4 Discretionary Access Controls

Developers shall construct their segments so that individual menu items and icons can be profiled through use of COE profiling software. The profiling software allows a site administrator to limit an individual operator's access to segment functions by menu item and by icon.

5.8.5 Command-Line Access

It is highly desirable for segments not to provide an xterm window or other access to a command-line. Segment features should be designed and implemented in such a way that operators are not required to interact with the application or operating system by entering commands in a command-line environment. Operators should interact with applications and the operating environment through graphical user interfaces.

Situations requiring superuser (i.e., root) command-line access shall require the operator to log in as a normal user then use the `su` command (for UNIX) to become a superuser. Superuser access by other means is not permitted unless the DII COE Chief Engineer grants prior authorization. Permission will be granted only for COE-component segments.

Segments that provide command-line access shall audit entry to and exit from the command-line access mode. Entry to command-line access mode shall require execution of the system login process so that the user is required to enter a password. For example, the UNIX command

```
xterm -exec login
```

will create an xterm window that requires the operator to provide a login account and password.

Segments which require command-line access shall use the `$CMDLINE` keyword (and the required `$KEY` keyword) in the Direct segment descriptor to document that the segment provides command-line access. If the segment provides superuser privileges, the `$SUPERUSER` keyword must also be stated in the `Direct` segment descriptor.

5.8.6 Privileged Processes

Segments shall minimize use of privileged processes (e.g., processes owned by root or executed with an effective root user id). In all cases, privileged processes shall terminate as soon as the task is completed. Privileged processes require prior Chief Engineer approval.

(UNIX) The names of the privileged processes must be listed in the `Processes` segment descriptor with the `$PRIVILEGED` keyword. The `$KEY` keyword must also be used to indicate that authorization has been granted by the Chief Engineer.

(UNIX) Shell scripts that `SUID` or `SGID` to root are strictly forbidden.

5.8.7 Installation Considerations

Segments shall not require `PostInstall`, `PreInstall`, or `DEINSTALL` to run with root privileges unless permission to do so is granted by the Chief Engineer.

Segments shall not alter the UNIX umask setting established by the COE.

5.8.8 File Permissions

Segments shall satisfy at least one of the following two requirements:

1. The segment contains only subdirectories directly underneath the segment's home directory. All files are at least one level down from the segment's home directory.

2. The segment has no directories or files that have the equivalent of the UNIX 777 file permissions.

This requirement is an attempt to provide a reasonable balance between security requirements and migration of legacy systems. The main issue is that files and directories should have read/write/execute permissions set for authorized, and only authorized, users.

Segments shall not place any temporary files in the directory pointed to by `TMPDIR` unless deletion, alteration, or examination of such files by another segment or user poses no security concerns.

5.8.9 Data Directories

Segments which contain data items with mixed permissions (e.g., some are read-only, some are write only, some are read/write) shall be split into separate directories underneath the segment's `data` subdirectory (for reasons explained in section 5.8). File permissions on the separate directories shall be set to prevent unauthorized access to data files. No file shall be "world writeable" (i.e., writeable by any user) unless authorized by the Chief Engineer.

5.9 Database Considerations

COE-based systems commonly make extensive use of databases. Database considerations are therefore of paramount importance in properly architecting and building a system. This section provides more detailed technical information on properly designing databases and database applications.

5.9.1 Database Segmentation Principles

A COE database server is a COTS DBMS product. It is used in common by multiple applications. It is a services segment and part of the COE. However, different sites need varying combinations of applications and databases. As a result, databases associated with applications cannot be included in the DBMS services segment. Instead, these component databases are provided in a database segment established by the developer. The applications themselves are in a software segment, also established by the developer, but separate from the database segment. If the data fill for the database contains classified data or is particularly large, that data fill must be in a separate data segment associated with the database segment.

5.9.1.1 Database Segments

The DBMS is provided as one or more COTS segments. These segments contain the DBMS executables, the core database configuration, database administration utilities, DBMS network executables (both server and client), and development tools provided by the DBMS vendor. Databases are provided as database segments. These segments contain the executables and scripts to create a database and tools to load data into the database.

The following functional groupings are used to provide database services. The configuration of COTS segments that provide them may vary depending on the DBMS and the specific configuration chosen. The COTS segments will usually be provided as a COTS DBMS server segment and a COTS DBMS client segment, installed on the database server platform and on the client platforms, respectively. Specific implementations of COTS DBMS segments are discussed in Appendix F.

1. **DBMS Server**. This functional group provides the DBMS executables, the DBMS's network services executables, and the core database. Its components are usually part of the DBMS server segment.

2. **DBMS Tools**. This functional group provides the executables for other DBMS applications (e.g. Oracle*Forms development tools). Its components are usually part of the DBMS server segment.

3. **DBMS DBA Tools**. This functional group provides the executables for tools used by database administrators (e.g. Oracle's ServerManager). Its components are usually part of the DBMS server segment, but may also be incorporated in the COTS DBMS client segment.

4. D**BMS Client Services**. This functional group provides the client network services for the DBMS and runtime executables for other DBMS applications (e.g. Oracle*Forms 4.0 `runform` executable). Its components are installed on the network's application server and on individual platforms.

The following specific segments are prepared by developers to provide databases within a COE-based system configuration.

1. **Application Database Segment**. This database segment contains a database belonging to a component application. It is installed on the database server.

2. **Application Client Segment**. This software segment contains applications that access a database created by an Application Database Segment. It is installed on the network's application server or on individual platforms.

3. **Application Database Data Segment**. This data segment contains the data fill of a component database when that data fill must be separated from the Application Database Segment. It is installed on the database server.

5.9.1.2 Database Segmentation Responsibilities

Three groups are involved in the implementation of database segments: DISA, the application developers, and the sites' database administrators. The developers and DISA work together to field databases and associated services for the DBAs to maintain. DISA provides the DBMS as part of the COE. Developers provide the component databases. Sites manage access and maintain the data. Users interact with the databases through mission applications and may, depending on the application, be responsible for the modification and maintenance of data in the databases.

5.9.1.2.1 DISA

DISA or the cognizant DOD Program Office provides the core database environment in which the applications' database segments will be integrated. The basic functionality provided with that core environment gets the database server ready for developers to add their databases and for the sites' database administrators to add and administer users.

The initial database contains the data dictionary, system workspace and recovery storage, storage for the database component of any vendor tools, and an initial allocation of user workspace and temporary storage. The application servers and client platforms are set up with the DBMS client environment so that users need only execute the environment shell script to be able to connect to the server. Finally, the initial operating system and DBMS accounts are established on the database server for the sites' database administrators.

5.9.1.2.2 Developers

Developers are responsible for providing everything associated with their application's database. Developers must define the owner account(s) for their base data objects. They must define and create the data objects within those owner accounts. Aside from the data proper, developers must determine and define the access levels and privileges that must exist for their segment's database. Database roles must be used to implement the users' access controls to ease the maintenance burden on the DBA.

- Developers may implement specific auditing within their applications and databases, but shall not modify the system's security audits.

- Developers shall provide scripts for the DBA's use to add, modify and remove user privileges.

5.9.1.2.3 Database Administrators

The System and Database Administrators at each site are responsible for creating, modifying, and removing users' DBMS and UNIX accounts using COE Tools. For security and ease of management, a "unitary login" or single account name for each user for both the operating system and the DBMS is being adopted for COE-based system. This means that users cannot use DBMS accounts defined by developers and that developers cannot assume the existence of any particular user accounts except for accounts created by the developer to support DBMS services. It also means, as required by the system Security Policy, that database actions can be traced to the individual user. Security auditing is the responsibility of the sites' DBAs. They are implemented as each site needs using the audit features provided by the DBMS.

A DBA creates users' DBMS accounts as part of the process of granting users access to applications and their associated databases. COE Tools are used to accomplish this. In order for these tools and the grants process to work properly and smoothly, the developers must provide procedures, scripts, and instructions

for the DBA's use. Users' access will change over time and few users will have access to all applications. The developers' procedures must support the addition of users and the revocation of users' privileges. Since those privileges correspond to applications or sets of applications, separate procedure scripts must be provided for each application or set. If an application has multiple levels of privileges, then multiple procedures must be provided.

5.9.1.3 DBMS Tuning and Customization

The core database server segment(s) is (are) configured and tuned by the organization responsible for it (e.g., DISA, GCCS, GCSS) based on the combined requirements[48] of all developers' databases (within the program or DOD wide) taken together. Developers provide these requirements during Segment Registration. This allows the DBMS Server segments to be reasonably independent of particular hardware configurations and ignorant of specific application sets. It is not tuned or optimized beyond that.

The final tuning of the DBMS cannot be accomplished until a complete configuration is built and it has an operational load. Developers should provide information into the tuning process, but should not make their applications dependent on particular tuning parameters. Where a non-standard parameter is required for operations, developers must provide that information to DISA so the DBMS services segment can be modified accordingly.

The developers need to communicate any design assumptions and DBMS configuration requirements that must be incorporated in the DBMS set-up. If, for example, developers need any settings in the Oracle `initDII.ora` file that are not the default settings for the current data server segment used in the currently available data server segment, that information needs to be provided to the DII COE Chief Engineer or responsible Program Chief Engineer early in the integration process for a forthcoming release. Based on the impact of the change, DISA or the responsible Program Office can decide to modify the baseline server configuration or to develop a database server patch segment to accompany the application's database segment and modify the in-place database server segment.

Similarly, sizing of system recovery logs, log archiving directories, and users temporary workspace is based on the combination of the requirements of the various applications that use DBMS services. Developers must communicate their minimum requirements for these so that the core DBMS is not set to be too small. Most of the application tools provided by DBMS vendors are incorporated in the DBMS segment in the functional category of Server Tools. To ensure that needed tools are available, developers should advise the Chief Engineer what COTS tools they intend to use when registering the segment. When such tools are used, the developer must identify the dependency under the database application segment's `Requires` descriptor.

- Developers shall not modify the core DBMS instance's configuration. Extensions or modifications of that configuration require the specific approval of the DII COE Chief Engineer and will be implemented by DISA in the COTS DBMS segment.

- If developers modify any of the executable tools (e.g. add User Exits to Oracle*Forms), then the modified version of the tool does not reside with the core database services, but becomes a part of the application's client segment. This prevents conflicts among different modified versions of a core function. The maintenance of that modified tool also becomes the responsibility of the developers.

5.9.2 Database Inter-Segment Dependencies

A key objective of the segmentation approach is to limit the interdependencies among segments. Ideally, database segments should not create data objects in any other schema or own data objects that are dependent on other schemas. However, one purpose in having a Database Server is to limit data

[48] An implication of this statement is that the combined requirements may lead to the need to develop a multiple instance database server segment.

redundancy and provide common shared data sets. This means that there will usually be some dependencies among the databases in the federation. This section addresses the management of such dependencies.

The following principles apply when inter-database dependencies exist:

- The database schema within a segment that will own the parent object will create that object.
- The database schema within a segment that will own the child (dependent) object will create that object.
- Database schemas with inter-database dependencies will strive to keep those dependencies in segments separate from the non-dependent portions of the schema.
- The referencing object, not the one that is referenced, owns referential dependencies (e.g. foreign keys). If the only dependencies are referential, separate segments are not needed.
- Schemas retain their autonomy. The developer of a dependency (including referential dependencies) is responsible for maintaining that dependency should other developers change their database schemas.

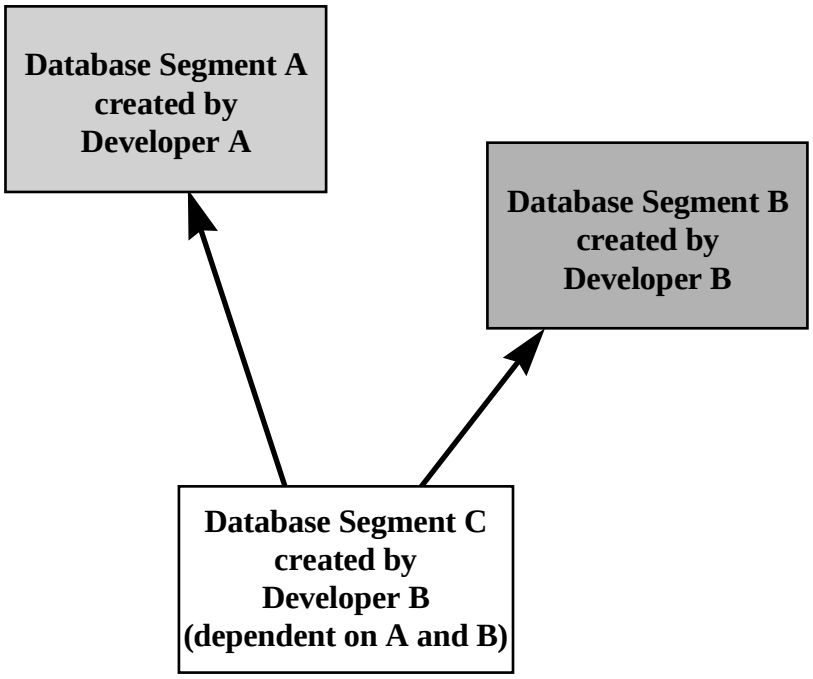
The following are general requirements for database segments.

- Application Database Segments shall not make modifications to another segment's application database. If a schema in an application database needs to create data objects in some schema belonging to another application database segment, those objects will be placed in the application database segment that owns those schema objects. Developers shall not create indexes on another application database segment's tables because of the performance problems they can cause.

- Developers will not modify the schema of another segment's database. If changes to table or column definitions are needed, they must be effected by the developer of the database.

- When dependencies exist they will be documented under the `Requires` descriptor of the `SegInfo` file. Object dependencies will be document under the `Database` descriptor of the `SegInfo` file.

The following example illustrates (see Figure 5-18) how dependencies are to be created and managed. The developers of database B need to attach a trigger to a table in database A. This trigger will feed data from A to B every time that table is modified. Rather than include the trigger as part of B's Database Segment, it is put into a separate Database Segment C, that modifies Database Segment A. C, the inter-database segment, is dependent on the prior installation of both database segments and is so labeled under its `Requires` descriptor. The table is listed in the `$MODIFIES` section of the `Database` descriptor.



Segment dependencies are listed in the Requires descriptor
Object dependencies are listed in the Database descriptor

Figure 5-18: Inter-Database Dependencies

5.9.3 Loading Data into Database Segments

After the objects belonging to a Database Segment have been created in `PostInstall`, they may need to be populated. Other objects, those containing dynamic data, may be initially empty. Where needed, a database segment can perform initial data fill in the Load Data phase of the `PostInstall`. Several methods are discussed below that can be used to accomplish data loads. Method selection should be based on the amount of data to be loaded.

If a small number of records are to be loaded into a table, the load can be accomplished with insert statements embedded in an SQL command script. The following excerpt is an example for loading data into Oracle.

```
sqlplus -silent DBSORT/${DBO_PWD} <<eof
INSERT INTO SORTSM_BIDES (UIC, SECUR, TIME,SCLAS)
      VALUES ('N00001','U',sysdate,'U');
INSERT INTO SORTSM_BIDES (UIC, SECUR, TIME,SCLAS)
      VALUES ('N00002','U',sysdate,'U');
INSERT INTO SORTSM_BIDES (UIC, SECUR, TIME,SCLAS)
      VALUES ('N00003','U',sysdate,'U');
eof
;;
```

If a large amount of data is to be loaded into a database table, the use of a data loading utility furnished by the RDBMS is usually more efficient. In this case, the utility can be invoked from the LOAD_DATA

section of the database definition script. Examples of these data loading utilities are Oracle SQL*Loader, Informix dbload, Oracle or Informix Import, and Sybase bcp. These utilities require that the data to be loaded be stored in a file with a specific format.

Files used for data fill belong in the `data` subdirectory of the database segment. The data directory within the segment can also be used as a 'mount point' for CDROM, tape drive, or other bulk storage devices. This is the preferred approach for large data loads. It allows the segment to be filled without occupying disk space during the data fill.

The security classification of the data to be loaded must be considered during the implementation of a database segment. When a classified data fill is part of the database segment, the entire segment becomes classified at the same level as the data. Therefore, developers must separate the data fill from the database segment when the database schema is not classified, but the contents are. The intent here is to keep database segments unclassified as much as possible so schemas can be reused. The security classification of a DII COE system (e.g. GCSS) is a separate issue and is addressed in the security policy of that system's program office.

If a separate data segment is provided to accompany a database segment, that data segment must have a `DEINSTALL` capability. This frees storage after the data fill is complete.

It can take a long time to fill a large database. Developers should indicate the approximate load time in their `ReleaseNotes`. The data load time can be reduced by loading the data before creating the database constraints and indexes. Estimating the load time should only be done with clean data that has been tested against the database constraints.

5.10 Tailoring the COE

Most properly designed segments will not require any extensions to the COE, except for the need to add icons and menu items. This subsection describes some of the more commonly required extensions, and techniques for addressing less frequently encountered extensions.

5.10.1 Adding Menu Items to the Desktop

Adding menu items is usually required only when installing a software segment. Two pieces of information are required: the name of the affected account group(s) and the menu items to add. Refer to the `SegName` and `Menus` descriptors.

The installation software appends the contents of the segment's menu files to the corresponding menu files in the affected account group(s). This forms a master template in the affected account group's `data/Menus` subdirectory that is subsequently used to create operator profiles. Segments should use the `APPEND` directive in the menu files to add items. Refer to the *Executive Manager Programmer's Guide* in the Developer's Toolkit documentation for the format of menu files.

Previous COE releases included a system menu bar that was displayed at the top of the screen, just below a security banner. The COE no longer automatically provides a system menu bar. Segments that require a system menu bar must use the Executive Manager APIs to explicitly add menu items when the application initializes. Developers may only add menu items that are contained within the current user's profile. The APIs are constructed to prevent addition of menu items to the system menu bar that are not contained in the current user profile.

Segments that use a system menu bar must also use the APIs to remove their system menu bar additions when the application terminates. Refer to the *User Interface Specification* for guidance on when it is appropriate to use a system menu bar versus desktop icons.

5.10.2 Adding Icons to the Desktop

As with menus, adding icons is usually required only for software segments. Two pieces of information are required: the name of the affected account group and the icons to add. Refer to the `SegName` and `Icons` descriptors above.

The installation software appends the contents of the segment's icon files to a master list located with affected account group(s). This forms a master template in the affected account group's `data/Icons` subdirectory that is subsequently used to create operator profiles. Refer to the Executive Manager API documentation for the format of icon files.

Refer to the *User Interface Specification* for guidance on when it is appropriate to use a system menu bar versus desktop icons.

5.10.3 Modifying Window Behavior (UNIX)

The *User Interface Specification* defines required window behavior for all segments. X Windows controls window behavior through a collection of resource definitions. The resource definitions consulted are as follows (if they exist):

1. Files located in the directory `/usr/lib/X11/app-defaults`.
2. Files in the directory pointed to by `XAPPLRESDIR`.
3. Resources inherited from the display's root window.
4. The file `$HOME/.Xdefaults`.

5. The file pointed to by XENVIRONMENT.

X Windows processes the controls in the order shown, and in such a way that the last control specified overrides any preceding controls.

The COE must carefully control resources to avoid conflicts between segments. Therefore, segments shall *not* place files in directories "owned" by X Windows (e.g., /usr/lib/X11/app-defaults.) Instead, segments shall place their resources in the subdirectory data/app-defaults underneath the segment directory as shown in Figure 5-2. At install time, the installation tools create a symbolic link underneath $DATA_DIR/app-defaults to each of the files contained in the segment. For this reason, segments must use their segment prefix to name all app-defaults used in this manner.

Figure 5-2 also shows that segments may place additional fonts underneath the segment's data/fonts subdirectory. At install time, the installation tools create a symbolic link underneath $DATA_DIR/fonts to point to each of these files. Segments shall use their segment prefix to name font files used in this way.

The COE establishes the setting for environment variables XFONTSDIR, XAPPLRESDIR, and XENVIRONMENT. Segments shall not modify their value. They are set as defined in subsection 5.3.

Motif follows a similar strategy for setting resources. The COE uses the Motif software provided with CDE software. Refer to the Developer's Toolkit documentation for more details on how Motif operates within the CDE environment.

Segments may *not* place files in any directory "owned" by Motif (e.g., /usr/lib/X11/app-defaults/Mwm) or CDE, nor may segments alter the account group's .mwmrc resource file, if it exists.

To summarize, for DII compliance:

- Segments shall *not* modify vendor distributed X Windows, Motif, or CDE system resources (Xdefaults, rgb.txt, etc.).
- Segments shall *not* place files in the X, Motif, or CDE distribution directories (e.g., /usr/lib/X11/app-defaults).
- Segments shall use the segment prefix to uniquely name files underneath the segment's data/fonts and data/app-defaults subdirectories.
- Segments shall *not* modify the COE established setting for XAPPLRESDIR, XENVIRONMENT, or XFONTSDIR.
- Segments shall *not* modify the affected account group's .mwmrc file, if one exists.

5.10.4 Using Environment Extension Files (UNIX)

The ReqrdScripts descriptor allows extensions to the affected account group's "dot" files (.cshrc, .login, etc.). This is most frequently done to add environment variables. However, unregulated use of environment variables is detrimental to the system. The amount of space the operating system reserves for environment variables is limited and loading a large number of segments could quickly exhaust this scare resource. Each time a process is spawned, the child process inherits environment variables from the parent. Resolving a large number of environment variables can take a significant amount of time and hence degrade system performance.

DII compliance requires adherence to the following guidelines:

- Do not include development environment variables in runtime environment scripts or extension files.
- Use “short names” for environment variables. UNIX stores environment variable names as character strings in the environment space, so the longer the variable name, the faster environment variable space is exhausted.
- Reuse environment variables already defined by the COE or affected account group.
- When feasible and efficient, use operating system services (such as pipes and streams) or data files to communicate with other segments, or between components within the same segment.
- Do not use environment variables to communicate control data between components within the same segment. Use operating system services or data files.
- Do not define environment variables that can be derived from other environment variables. For example, to define `MYSEG_BIN` through

```
setenv MYSEG_HOME       /h/MySeg
setenv MYSEG_BIN        $MYSEG_HOME/bin
```

wastes environment variable space. The COE guarantees a predictable directory structure, so `$MYSEG_HOME/bin` can be used directly instead of `$MYSEG_BIN`.

- When feasible, have segment components create environment variables once they begin executing through `putenv` or through “sourcing” a file containing needed environment variables. This approach ensures that segment-specific environment variables are inherited locally by a single segment, not globally by all segments.

5.10.5 Using Community Files

Community files are any files that reside outside a segment’s assigned directory. (Data files owned by the segment underneath `/h/data` are considered an exception.) Most required community file modifications are handled automatically by the installation software through descriptor directory files. The `Community` descriptor is used when the installation software cannot provide the modifications required.

All community file modifications are carefully scrutinized at integration time because of the potential for conflict with other segments or the runtime environment. Developers should seek guidance from the Chief Engineer before modifying any COTS community files (those owned by UNIX, X Windows, Motif, Oracle, Sybase, etc.).

5.10.6 Defining Background Processes

When an operator logs in, the operating system uses various files to establish a runtime environment context. Segments use the `Processes` descriptor to add other background processes to the runtime environment.

The COE differentiates between eight different types of processes:

Boot	Processes launched each time the computer is booted or rebooted. Designate boot processes with the `$BOOT` keyword.
DCE Boot	DCE processes launched each time the computer is booted or rebooted. Designate DCE boot processes with the `$DCEBOOT` keyword.

RunOnce Processes launched the next time the computer is rebooted. These are "one-shot" processes and are only run the next time the computer is rebooted, but not for reboots thereafter. Designate RunOnce processes with the `$RUN_ONCE` keyword.

Periodic Processes launched at boot time that automatically run periodically at specified intervals (e.g., 6 hrs, 24 hrs) with no other user actions required to initiate the process. These processes are equivalent to UNIX `cron` process. Use the `$PERIODIC` keyword to indicate these types of processes.

Privileged Processes that require "superuser" privileges to execute. Use the `$PRIVILEGED` keyword to indicate these type of processes.

Background Processes launched the first time an operator logs in after a reboot; these processes remain active in the background even after the operator logs out. Designate background process with the `$BACKGROUND` keyword.

Session Processes launched when an operator logs in and remaining active only while the operator is logged in. Designate session processes with the `$SESSION` keyword.

Transient Processes launched in response to operator selections from an icon or menu. Transient processes typically display a window on the screen, perform some specific function in response to operator actions, and then terminate. In some cases, the processes spawned may stay active for the length of the session, but in all cases, the Executive Manager terminates transient processes when the operator logs out. Designate transient processes through the `Menus` and `Icons` descriptors.

Note: Because of the potential impact to other segments, system performance, and system integrity, all processes except Session, and Transient processes require prior approval by the Chief Engineer. Boot, DCE Boot, privileged, and periodic processes are *strongly* discouraged.

5.10.7 Reserving Disk Space

Segments frequently require additional disk space to accommodate growth over time as the system operates. For example, communications logs are empty when the system is initially installed, but will occupy space as messages are received and logged. Segments may reserve additional disk space through the `Hardware` descriptor.

The installation software keeps track of how much disk space is actually in use and how much is reserved. A segment will not be installed if the amount of space it occupies, plus any space it reserves, exceeds the amount of unreserved disk space. The installation software allows the operator to select how full the disk can be (80, 85, 90, or 95% of capacity). These safeguards are in place to avoid filling up the disk, but segments are responsible for detecting when the amount of space requested is not available.

In rare situations, segments may require space on multiple disk partitions. See the `$PARTITIONS` keyword for the `Hardware` descriptor.

5.10.8 Using Temporary Disk Space

Segments may require temporary disk space during segment installation and during system operation. The COE provides techniques for accommodating both uses for temporary space.

Temporary disk space may be requested during segment installation through the `$TEMPSPACE` keyword in the `Hardware` descriptor. The installation software sets the `COE_TMPSPACE` environment variable to point to the location where temporary space is allocated. This environment variable is defined *only* during segment installation. The installation software automatically deletes all files in this temporary area when segment installation is completed.

The environment variable `TMPDIR` points to a temporary directory that may be used during system operation. However, there is a limited amount of disk space set aside for temporary storage so it must be used sparingly. A better approach is for segments to store temporary data in their own `data` subdirectory.

Segments that use `TMPDIR` must delete temporary files when they are no longer required. For UNIX systems, all files in this directory are automatically deleted when the system is rebooted. This is *not* true for NT platforms. All segments, as a matter of good programming practices, should delete temporary files when they are no longer needed.

5.10.9 Defining Sockets

Requests to modify the `/etc/services` file to add sockets is done through the `COEServices` descriptor. This control point for requests to add socket names and ports helps avoid conflicts between segments. Port numbers in the range 2000-2999 are reserved for COE segments. Segments should avoid creating sockets with port numbers less than 1000 since these are generally reserved for operating system usage.

5.11 Miscellaneous Topics

This subsection discusses a variety of miscellaneous topics that are related to segmentation, use of the DII COE, etc.

5.11.1 Color Table Usage

The COE must carefully control how the color table is used to avoid objectionable “false color” patterns that may appear when mouse focus changes from one window to another. The *User Interface Specification* gives guidance on what colors to use from a human factors perspective, but it does not provide guidance on how segments are to coordinate such usage through the COE.

This document will be expanded to include guidance for color table usage as the impact of COTS products and legacy applications is evaluated.

5.11.2 Shared Libraries

The COE strongly encourages the use of shared libraries to reduce memory requirements. Developers may create shared libraries (DLLs for NT platforms) through use of the `SharedFile` segment descriptor.

(UNIX) Developers should also link to X and Motif shared libraries to reduce memory requirements. The Motif libraries provided by CDE should be used instead of the libraries provided by Motif or some other source. This alleviates the need to maintain Motif shared libraries used both by the desktop (e.g., CDE) and other applications.

5.11.3 Adding Network Host Table Entries

Platform IP addresses and hostnames are site-dependent. Hostnames in particular are most often selected by the site and usually cannot be predicted in advance. Therefore, segments shall not include any assumptions about a platform having a specific name or following any particular naming convention, nor make any assumptions about a specific IP address class.

Segments should rarely need to add entries to the network host table. An operator usually establishes such entries through system administration functions. For those situations where a segment must do so, the `$HOSTS` keyword in the `Network` descriptor allows IP addresses, hostnames, and aliases to be added to the network host table. The address may be added to either the local host table, or to the DNS/NIS/NIS+ maintained host table.

Prior permission must be given by the DII COE Chief Engineer to use the `$HOSTS` keyword, and permission will be granted only for COE-component segments. `VerifySeg` will issue a warning for any segment which uses the `$HOSTS` keyword, and a warning if the segment does not include the `$KEY` keyword. A future release will issue an error if the segment does not provide a valid authorization key.

5.11.4 Registering Servers

Servers are registered with the COE through the `$SERVERS` keyword in the `Network` descriptor. Only COE-component segments may register servers. Prior permission must be given by the DII COE Chief Engineer to use the `$SERVERS` keyword. `VerifySeg` will issue a warning for any segment which uses the `$SERVERS` keyword and strictly fail the segment if it is not a COE-component segment.

A segment that needs to determine the location of a server may use the `COEFindServer` function (see Appendix C).

5.11.5 Adding and Deleting User Accounts

Segments are not normally allowed to create operator accounts (e.g., UNIX user login accounts). Segments may create system accounts, through the `COEServices` descriptor, for the purpose of establishing file ownership. Operator accounts are normally added to the system through use of the Security Administrator application. They are customizable by security classification level, by access permissions granted or denied against application objects, and by granting or denying access to menu or icon items. The segment descriptors `AcctGroup`, `Security`, `Permissions`, `Menus`, and `Icons` provide these controls.

Figure 5-3 shows that operator accounts may be global or local. This attribute is specified when the operator account is created. If the server that contains operator accounts is down, global operator logins will be unavailable until the server is restored.

Profiles may also be global or local. This attribute is determined when the profile is created. If a global profile is not available at login time (e.g., the server is down), login proceeds but the operator is notified of the problem and the system is placed in a safe state.

Some segments require the ability to perform additional operations when a user account is created, or to perform cleanup operations when a user account is deleted. This is done by using the `$ACCTADD` and `$ACCTDEL` keywords in the `Direct` descriptor. Moreover, the `$PROFADD`, `$PROFDEL`, and `$PROFSWITCH` can be used to perform segment-dependent operations when user profiles are created or deleted, or when a user switches from one profile to another. Due to security implications, these keywords require prior permission from the Chief Engineer and use of the `$KEY` keyword.

5.11.6 Character-Based Applications

Support for character-based interfaces is provided through the `CharIF` account group. An account is established for individual users through the same process as all other accounts, but the account is identified as a character-based interface account only. Operator profiles may be set up, but only those segments that support a character-based interface (see the `Direct` descriptor) are accessible.

The remote user connects to the designated server through a remote login session. Once connected, the user is prompted for a login account and password. A menu of options, such as

```
0) Exit
1) AdHoc Query
2) TPFDD Edit

Enter Option:
```

is presented to the user. The option selected is executed and results are displayed on the user's remote, character-based display.

5.11.7 License Management

The COE contains a license manager to administer COTS licenses. Vendors take a variety of approaches in how they control and administer licenses. For this reason, the techniques for automating license management are still under development and are being handled manually. Refer to the DII COE Chief Engineer for further assistance in creating a segment that requires a license manager.

Developers should include the COTS vendor's version number as the secondary version number as described in Chapter 2. This will facilitate automated license management.

5.11.8 Remote versus Local Segment Execution

Segments which are remotely launchable are designated by the `$REMOTE` keyword in the `Direct` descriptor. This feature is not currently implemented, but is reserved for future implementation. Developers are encouraged to use the `$REMOTE` keyword and design their segments to account for local versus remote execution. Thus, when this feature is fully implemented, developer segments will be positioned to take advantage of the capability.

5.11.9 Modifying Network Configuration Files

Setting up a network requires modification of several network configuration files to set netmasks, identify subnets and routers, etc. Proper network configuration is essential for proper system operation and performance. For this reason, only COE-component segments may establish network configuration parameters. This is accomplished through the `Network` descriptor.

Prior approval from the DII COE Chief Engineer is required. `VerifySeg` will issue a warning for any segment that uses the `Network` descriptor and strictly fail the segment if it is not a COE-component segment. Note that the `$KEY` keyword must also be specified to give a valid authorization key.

5.11.10 Establishing NFS Mount Points

NFS mount points are defined through the `$MOUNT` keyword in the `Network` descriptor. Establishing mounted file systems can seriously degrade system performance. Poor design choices that result in several different mount points can create single points of failure, or result in sequencing problems when the system is loaded or rebooted. For these reasons the DII COE Chief Engineer must approve mount points for COE-component segments. The cognizant Chief Engineer must approve mount points for mission application segments.

`VerifySeg` will issue a warning for any segment which uses the `$MOUNT` keyword. It will strictly fail any COE-component segment that does not specify the `$KEY` keyword.

6. PC-Based Applications

This chapter describes the DII COE features that are available for PC platforms. The present DII COE supports PC Windows NT[49] only. The COE concept is not specific to UNIX, or NT, or any other operating system or windowing environment. However, certain adjustments to COE implementation details are required to support differences between the PC-based NT environment (use of '\' versus '/' in naming directories, byte swapping, etc.) and UNIX, as well as to take advantage of features offered in the NT environment (e.g., registry).

The extensions described in this chapter to accommodate NT are not platform-dependent (e.g., limited to 80x86 PCs). Commercial industry has implemented the Microsoft NT operating system on selected other platforms (e.g., DEC), but such platforms are not presently in wide use in the DII community. COE support for NT on platforms other than PCs will be considered when they are in widespread use in the DII community. Throughout this version of the *I&RTS*, NT and PC may be used interchangeably with the understanding that NT is not limited to PC platforms.

[49] Windows 3.1 and Windows for Workgroups 3.11 are not supported. Windows 95 is not presently a supported platform because of known security problems within the operating system. When the security problems are resolved, Windows 95 may be added to the list of supported platforms.

6.1 Disk Directory Structure

The NT-based COE uses the same basic directory structure shown in related figures from Chapter 5. However, Intel-based computers store data bytes in a different order than other processors. This makes data sharing via disk more difficult. This section describes the COE disk directory extensions required to support PCs.

Basic Directory Structure

The logical directory structure shown in Chapter 5 is preserved for PCs. On the primary disk drive, subdirectory `\h` is created at the root level with subdirectories `COTS`, `AcctGrps`, `COE`, `data`, etc. Unless overridden by the installer, the installation software will attempt to put segments on the primary disk drive first. If it cannot do so, it will load the segment on the next available hard disk. The environment variable `INSTALL_DIR` is set to point to where the segment was loaded at install time, just as for UNIX platforms, and includes the disk drive designation in the pathname.

Segment Directory Structure

A `Scripts` subdirectory is optional for NT segments because environment extension files are not supported, nor are they needed. Account group segments that need to establish global environment settings shall do so by entering required settings in the registry. Segments that need to establish local environment settings may do so through a `.INI` file that shall be located in the segment's `data\INI` subdirectory. All of a segment's private INI files shall be stored in the segment's `data\INI` subdirectory.

NT segments shall place all executables in the `bin` subdirectory. Segments that contain dynamic link libraries (DLL files) shall place them in the `bin` subdirectory. Except for COTS segments, segments are not allowed to load DLL files in any other subdirectory.

USERS Directory Structure

The NT COE uses the same operator directory structure as the UNIX COE, as described in Chapter 5. Local operator accounts are specific to a single NT platform, while global operator accounts are accessible from any NT PC on the network. However, operator accounts may not be mixed between UNIX and NT platforms. Thus, an operator account, whether global or local, is either an NT operator account or a UNIX operator account, but never both.

Global operator account subdirectories (e.g., `\h\USERS\global`) are physically located on an NT designated as the server. This directory is made accessible to other PCs on the network through the `share` command.

Environment variables `USER_HOME`, `USER_DATA`, and `USER_PROFILE` are set by the appropriate account group and have the meaning described in Chapter 5. They are provided for backwards compatibility and should not be used in the NT-based COE. As with UNIX applications, segments shall use a *Preferences* API to locate user-related data. This is because data may ultimately be moved to the registry or reside in different locations depending upon the NT configuration (e.g., workgroups versus domains). By using the *Preferences* APIs, the developer can assure future compatibility.

Data Directory Structure

Chapter 5 defines data in terms of data scope. Local data is stored underneath `\h\data\local` while global data is stored underneath `\h\data\global`. Because data stored on the PC is not directly compatible with UNIX platforms, an additional data subdirectory is created for storing PC *only* global data. This is the subdirectory `\h\data\PCglobal`. Segments shall follow the same rules for this directory as for the `\h\data\global` directory, except that only PC segments are allowed to access it. This

subdirectory is physically located on a PC designated as the server and made accessible to other PC platforms on the network.

Like global data, `PCglobal` data is shared between platforms. However, `PCglobal` data (and local data on PC platforms) is stored in native PC-byte order and can only be shared among PCs. PCs may also access data stored in the `\h\data\global` subdirectory. However, this directory is *always* physically located on a UNIX machine designated as a server. PC segments shall read and write data in the `\h\data\global` directory in network byte order. PC segments shall read and write data in the `\h\data\local` and `\h\data\PCglobal` directories in native PC byte order.

Miscellaneous

1. Segments shall use file extensions that correspond to conventional Windows usage. That is, use `.EXE` for executables, `.DLL` for dynamic link libraries, `.TXT` for ASCII text files, etc. Note this means that NT segment descriptor files **should** use the `.TXT` extension,[50] but **shall** use the `.BAT` or `.CMD` (for batch[51] files), or `.EXE` (for compiled programs) extension for `PostInstall`, `DEINSTALL`, `PreInstall`, and `PreMakeInst`.

2. Segments, excepting COTS segments and in some cases shared DLLs, shall not set the Windows `path` environment variable. If the segment provides shared DLLs for use by other software, and if there is no alternative way for that software to locate the DLLs, the segment may add a directory to the path for those DLLs.

3. Segments shall use the standard Windows APIs to locate a directory for temporary disk storage. This corresponds to using `/tmp` in UNIX. Segments shall delete temporary files when an application terminates. Unlike the UNIX-based COE, the NT-based COE does *not* automatically delete files in the Windows temporary directory when the computer is rebooted. This is in keeping with current commercial usage of the Windows temporary directory.

4. Segments shall not add a global "home" environment variable to the affected account group.

5. Environment extension files are neither supported nor required in the NT-based COE.

6. `app-defaults` subdirectories are not meaningful in the NT-based COE. Special handling of fonts (i.e., a `fonts` subdirectory) is not currently supported in the NT-based COE, but may be in the future. NT segments should not include either of these subdirectories. If they are included with a segment, the installation tools will not do any special processing for these subdirectories as they do for the UNIX-based COE.

[50] For backwards compatibility, NT segments may omit the `.TXT` extension. However, this is strongly discouraged. The segment must be consistent in either *always* using the `.TXT` extension or *never* using it. `VerifySeg` will strictly fail a segment that does not follow this convention. Otherwise it will be confusing and unclear which descriptor takes precedence when a segment includes the same segment descriptor, once with the `.TXT` extension and once without it.

[51] Developers should avoid the use of batch files and use executables whenever possible. Batch files, in PC NT, will cause a command shell window to pop up while the batch file is running.

6.2 Account Groups

Account groups in the NT-based COE correspond to Windows Program Groups. The present NT COE does not include the `CharIF` or `DBAdm` account groups.

When the COE is loaded, the installation tools create program groups `SecAdm` and `SysAdm`. The program items in each program group are determined as segments are loaded. Some program items, specifically for `SecAdm` and `SysAdm`, are provided by native Windows software and therefore will also be found in other program groups provided by Windows. This is done by creating duplicate icons that point to the same executable, not by creating multiple copies of the software.

As with the UNIX COE, the specific icons and program groups available to an operator depend upon the operator profile.

6.3 Registry Usage

Microsoft Windows programs have traditionally created "INI" files to store configuration information. Windows 95 and Windows NT use a *registry*[52] instead to store hardware parameters, configuration data, and Windows-maintained operator preferences. The registry is structured as a hierarchical database of keys organized into a tree structure.

NT segments should not overuse the Windows registry in place of INI files. In particular, operator preferences that are very segment specific should not be stored in the registry since this may needlessly fill up the registry, and it will be difficult to manage as user accounts are created and removed. Moreover, the registry is not portable between NT and UNIX. It is recommended that operator preferences be stored underneath `\h\USERS` to minimize porting problems between UNIX and NT applications. (Use the appropriate COE APIs to determine the correct data directory for the current operator.) Segments may use private INI files but, if they are used, they shall be located in the segment's `data\INI` subdirectory.

Except for COTS segments, segments shall not create root keys, but may create subkeys underneath the root keys as desired. In all cases, segments shall create segment subkeys underneath

```
HKEY_LOCAL_MACHINE\SOFTWARE\COE
```

using the convention *`SegType`*`\`*`SegDirName`* where *`SegType`* is one of the following:

`Account Groups`	for account group segments
`COE`	for COE-component segments
`COTS`	for COTS products
`Data`	for data segments
`Patches`	for patch segments
`Software`	for all other segment types.

`SegDirName` is the segment's directory name. Segments shall use the segment prefix to name all registry subkey entries.

For example, assume a software segment whose directory is `SegA` has a segment prefix `SEGA`. Assume the segment needs to store two pieces of information underneath `HKEY_LOCAL_MACHINE\SOFTWARE`:

1. the last coordinate system used (Universal Transverse Mercator [UTM], Lat/Long, etc.) and
2. the last time a certain parameter was computed.

Then the required registry path is

```
HKEY_LOCAL_MACHINE\SOFTWARE\COE\Software\SegA
```

and two appropriately named subkeys underneath this entry for storing value entries are `SEGA_Last_Coord` and `SEGA_Last_Time`.

> **Note:** The key `HKEY_LOCAL_MACHINE\SOFTWARE\COE` is created when the DII COE is installed.

Microsoft encourages use of the registry in some ways that are strictly forbidden in the COE because the `COEInstaller` tool performs some of these actions automatically. Segments, excepting COTS segments, shall not use the registry to duplicate any actions performed by the COE installation software:

[52] Developers should avoid overuse of the NT registry. It is best used for system-level constructs and *not* as a total replacement for `.INI` files.

- Segments shall *not* register "uninstall" information in the `Uninstall` key beneath `CurrentVersion`, with two exceptions: (1) when the segment is a COTS product that does register "uninstall" information as part of its setup, or (2) as authorized by the DII COE Chief Engineer. If the segment does register "uninstall" information, it shall specify the `$USES_UNINSTALL` keyword in the `Direct` descriptor.

- Segments shall use the `Processes` descriptor to specify background processes. Segments shall *not* add values to either the `Run` or `RunOnce` keys beneath the `CurrentVersion` key. The segment shall use the `$RUN_ONCE` keyword to specify the requirement to run certain executables the next time, and only the next time, the system is restarted. Use of this keyword requires approval by the cognizant DOD Chief Engineer.

6.4 Reserved Prefixes, Symbols, and Files

The segment prefixes listed as reserved in Chapter 5 are also reserved in the NT-based COE. The following segment prefixes are reserved and are specific to the NT-based COE:

NT	Generic NT segments
WIN	Generic Windows segments
WIN95	Windows 95 segments
WINNT	Windows NT segment for 80x86 platforms

The environment variables listed as reserved in Chapter 5 are also reserved in the NT-based COE. Segments shall not create environment variables with the same name as any reserved environment variable. The following have no meaning in the NT-based COE, and are not guaranteed to be set:

```
DISPLAY
LD_LIBRARY_PATH
SHELL
TERM
TZ
XAPPLRESDIR
XENVIRONMENT
XFONTSDIR
```

All remaining environment variables listed in Chapter 5 are also defined for the NT-based COE.

The root-level `AUTOEXEC.BAT`, `CONFIG.SYS`, `AUTOEXEC.NT`, and `CONFIG.NT` files are reserved files and shall not be modified by any segment, excepting COTS segments. Moreover, all windows INI files (specifically, `WIN.INI` and `SYSTEM.INI`) are reserved files and shall not be modified by any segment, excepting COTS segments. Segments should create and modify their own local INI files.

6.5 Programming Standards

Programming in the Windows environment is considerably different from the UNIX/X Windows environment. This subsection details programming practices that are required to minimize problems in mixing the two environments.

6.5.1 File System

Windows NT supports five file systems: FAT, VFAT, HPFS, NTFS, and CDFS. FAT (File Allocation Table) is the file system used by MS-DOS, but it is extended in both Windows 95 and Windows NT (version 3.5 and later) to support long filenames (e.g., VFAT). HPFS (High Performance File System) originated with OS/2®. NTFS (NT File System) originated with Windows NT as an improvement over both HPFS and FAT. CDFS (CDROM File System) is specific to CDROM devices.

NTFS is the file system required for the DII COE because its security architecture corrects known problems in FAT. DII-compliant systems shall be formatted to use NTFS. However, the FAT and VFAT file systems are the only available file systems for floppy disks. Therefore, the COE requires NTFS for hard disk drives, but supports FAT and VFAT for floppy drives. The type of file system in use should be transparent to most segments. When there is a choice, NTFS shall be used for hard and VFAT shall be used for floppy drives.

A further complication is that NTFS filenames use the 16-bit *Unicode*® character set instead of 8-bit ASCII. Unicode is a technique for representing foreign alphabets (Japanese kanji, Chinese bopomofo, Greek, etc.). NT segments are not required to create Unicode strings, but segments must be able to read filenames that may be Unicode strings. This requirement is necessary because commercial products may be distributed on media that use Unicode filenames and because Windows NT uses Unicode strings internally.

Pathnames in Windows usually include a disk drive designation (e.g., `C:`). The disk drive containing the desired file may be located remotely on another machine. Windows allows symbolic names, called the *Universal Naming Convention* (UNC), to be given to remote paths so that an application need not know the platform, disk drive, or exact path to reach a particular file. UNC pathnames start with two backslashes (\\) followed by the server name, followed by the desired pathname and filename. Segments shall support the use of UNC pathnames.

To summarize,

1. Segments shall support the use of long filenames. Filenames are not allowed to contain embedded spaces and should use file extensions as appropriate to conform to standard Windows usage.

2. Segments shall support use of UNC filenames.

3. Segments shall be capable of correctly interpreting Unicode strings, those representing filenames.

6.5.2 Dynamic Link Libraries

NT segments shall use DLLs to the maximum extent feasible. DLLs are located in the segment's `bin` subdirectory, except for COE segments. COE DLLs are located underneath the directory `\h\COE\bin` for all COE segments.

Windows originally exported DLL functions by assigning ordinal numbers to each exported function. Modules linked to DLL functions by ordinal number. However, later versions allowed linking to be by symbolic name rather than ordinal numbers. All NT segments shall link by symbolic name and shall export DLL functions by symbolic name rather than ordinal numbers. The reason for this requirement is that ordinal numbers for exported functions could change with time, whereas symbolic names will not.

6.5.3 Graphics

PC segments shall support Video Graphics Adapter (VGA) and Super Video Graphics Adapter (SVGA) resolutions, and should use the Win32 API Graphics Display Interface (GDI) for creation of 2D graphics. This interface handles all calls made by applications for graphic operations and thus provides a standard interface for such calls. As a result, the Win32 GDI allows segments to be developed which are independent of the type of graphics output device in the end user's system. That is, segments need only make calls to standard graphic services provided by the Win32 subsystem regardless of the display, printer, or multi-media hardware used in the system.

To improve 2D graphics performance, the WinG library may be used. WinG is an optimized library designed to enable high-performance graphics techniques under Win32, Windows NT, Windows 95, and future Windows releases. Segments should use OpenGL APIs for 3D graphics. OpenGL is a software interface that allows the creation of high-quality 3D color images complete with shading, lighting, and other effects. OpenGL is an open standard designed to run on a variety of computers and a variety of operating systems. It consists of a library of API functions for performing 3D drawing and rendering.

6.5.4 Fonts

Windows supports three different kinds of font technologies to display and print text: raster, vector, and TrueType®. The differences between these fonts reflect the way that the *glyph* for each character or symbol is stored in the respective font resource file. In raster fonts, a glyph is a bitmap that Windows uses to draw a single character or symbol in the font. In vector fonts, a glyph is a collection of line endpoints that define the line segments Windows uses to draw a character or symbol in the font. In TrueType fonts, a glyph is a collection of line and curve commands as well as a collection of hints. Windows uses the line and curve commands to define the outline of the bitmap for a character or symbol in the TrueType font. Windows uses the hints to adjust the length of the lines and shapes of the curves used to draw the character or symbol. These hints and the respective adjustments are based on the amount of scaling used to reduce or increase the size of the bitmap.

Vector and TrueType fonts are device independent, while raster fonts are not. TrueType fonts provide both relatively fast drawing speed and true device independence. By using the hints associated with a glyph, application software can scale the characters from a TrueType font up or down and still maintain their original shape. Segments shall use TrueType fonts to take advantage of the increased performance, flexibility, and What-You-See-Is-What-You-Get (WYSIWYG) screen-to-printer characteristics. Customized application-specific fonts shall be avoided in favor of using industry standard fonts wherever possible.

6.5.5 Printing

NT segments shall use the built in printing facilities provided by Windows. This includes using the Windows supplied printer common dialog box for configuring a printer, selecting print quality, selecting the number of copies, etc. All access to the printer shall be through Windows APIs.

Developers should be aware that some Win32 APIs are available only in Windows NT. Developers may use these APIs, but should ensure that the segment still operates correctly in a Windows NT environment. As appropriate, NT segments should support drag-and-drop printing.

6.5.6 Network Considerations

UNC Filenames

NT segments shall support UNC filenames to access network shared drives and directories. If necessary, a segment can use the WinNet APIs to determine if a pathname is a network pathname.

The COE contains three pre-defined shared directories: `\h\data\PCglobal`, `\h\data\global`, and `\h\USERS\global`. The proper UNC filename to use for these three directories is determined by accessing registry subkeys underneath `HKEY_LOCAL_MACHINE\HARDWARE` as follows:

`COE\Shared\data_PCglobal`	`\h\data\PCglobal`
`COE\Shared\data_global`	`\h\data\global`
`COE\Shared\USERS_global`	`\h\USERS\global`

NT segments that create network sharable services or devices shall store UNC information in the registry. The subkey shall be either `COE\Shared` or `SEGS\Shared` depending upon segment type. The subkey shall be located underneath `HKEY_LOCAL_MACHINE\HARDWARE` for hardware devices (e.g., disk drives) or `HKEY_LOCAL_MACHINE\SOFTWARE` for software (e.g., servers). The segment shall document the proper registry information in the API documentation for the segment.

Network Byte Ordering

Computer architectures sometimes differ in the convention they use for how bytes are ordered in a word. This is the so-called "*big-endian, little-endian*" problem. Computers in which the *most* significant byte in a word is the leftmost byte use big-endian byte ordering. Computers in which the *least* significant byte in a word is the leftmost byte use little-endian byte ordering. Intel architectures use little endian byte ordering. When data is sent across the network, it is important to agree upon the same convention for byte ordering. The big-endian convention is also known as the *network byte order* and has been established as the industry standard.

The COE adopts the industry standard for byte ordering[53] and requires the use of network byte order for any data transmitted across a heterogeneous LAN. Segments shall ensure that all network data is transmitted in network byte order, except for certain data accessed on a PC-only network shared disk drive such as the `PCglobal` data directory. Segments shall use APIs in the WinSock interface to ensure that data sent across the network is in network byte order. Segments shall store disk data accessible only by PCs in native PC byte order, but shall store disk data accessible by non-PCs in network byte order. The shared data directories and byte ordering are as follows:

`\h\data\PCglobal`	PC native byte order. Data here is shared, but is restricted to only PCs.
`\h\data\global`	Network byte order. Data in this directory may be accessible from a UNIX platform as well as PCs.
`\h\USERS\PC`	PC native byte order. Data located here is specific to operator login accounts. Since a login account is either for UNIX or a PC but never both, this data is platform-specific.

Network Communications

Windows NT supports four transport layer protocols:

NetBEUI	provides compatibility with existing LAN Manager, LAN Server, and MS-Net installations.
TCP/IP	provides compatibility with standard UNIX environments and a routable protocol for wide area networks.

[53] DCE developers should use DCE functions to implement network byte ordering. All other developers should use XDR protocol.

Data Link Control (DLC)	provides an interface for access to mainframes and printers attached to networks.
AppleTalk®	provides interoperability with Macintosh networks.

TCP/IP is the COE standard network protocol. Segments shall perform network communications through WinSock APIs. Communications shall be designed to operate asynchronously to ensure that the server or application does not "hang" while waiting for a response.

6.5.7 Miscellaneous

The following statements apply to all new segment development. COTS segments may not meet all mandatory requirements, but shall be documented where they do not fulfill a mandatory requirement. To the extent possible, segments should conform to the requirements stipulated by Microsoft for allowing an application to use the Windows Logo. The *I&RTS* fully supports the Microsoft Logo branding approach as a subset of the requirements for full DII COE compliance.

Mandatory

1. All hardware shall be NT-compliant, as defined by the document *Microsoft Windows NT Hardware Compatibility List #4094*.

2. Segments shall support VGA and SVGA graphics.

3. Segments shall be "close aware." This means that the segment must enable the Close command and periodically check the close flag through the Query Close function.

4. Segments shall use common control and common dialog functions contained in `COMCTL32.DLL` and `COMDLG32.DLL`.

5. As appropriate, segments shall support cut and paste operations through the clipboard.

6. As appropriate, segments shall support drag and drop operations.

7. Segments shall support 16x16, 32x32, and 64x64 icons.

8. Segments shall *not* use MS-DOS APIs inside a compiled program. These functions are typically interrupt-driven or depended upon specific memory addresses and are not portable. Win32 APIs only are to be used within a compiled program. Segments may use MS-DOS commands within the various installation-related batch files.

9. Segments shall use *only* Win32 APIs. Win16 APIs are not supported and shall not be used unless they are part of a COTS product for which there is no 32-bit alternative.

10. Segments shall not duplicate functionality already provided by Windows.

11. Segments shall support long filenames and UNC.

12. Segments shall support the use of Unicode strings.

Optional

1. Segment developers should run the Windows SDK tool `PORTTOOL.EXE` to identify potential problems with how Windows APIs are being used.

2. Segments should operate under both Windows NT and Windows 95. The segment should degrade gracefully if it uses APIs found only in Windows 95 while running in a Windows NT environment, and vice versa.

3. Segments should define the `STRICT` constant when compiling Windows code. This enables strict type checking during compilation.

4. Segments should avoid using environment variables. The registry or local INI files are preferred alternatives.

5. Developers are encouraged to use message crackers contained in `WINDOWSX.H`. Message crackers are a set of macros that make code more readable, simplify porting, and reduce the need to do type casting.

6. As appropriate, segments should register icons for document types and provide a viewer to allow the shell to display them. This is done through the `HKEY_CLASSES_ROOT` registry. Refer to Microsoft documentation for the required procedures. A future COE release may provide segment descriptors to accomplish this.

6.6 Segment Installation

Segment installation follows the same sequence as for the UNIX environment, and is defined in Chapter 5. The key

```
HKEY_LOCAL_MACHINE\SOFTWARE\COE
```

is automatically created when the DII COE kernel is loaded. As segments are installed on the NT platform, `COEInstaller` creates registry entries under this key corresponding to segment type as explained in subsection 6.3. That is, assuming *`SegDir`* is the segment's directory name and *`SegType`* is the segment's type, the installer creates the following registry key entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\COE\SegType\SegDir
```

All entries underneath this registry key are deleted automatically when the segment is deleted.

`COEInstaller` sets the environment variables `INSTALL_DIR`, `MACHINE_CPU`, and `MACHINE_OS` for use in the `PreInstall.BAT` (or `.EXE`) and `PostInstall.BAT` (or `.EXE`) descriptors. `SYSTEM_ROOT` is set to indicate where Windows was installed. The installer also stores the location where the segment was loaded in the subkey *`SegDir`*`\SegPath`. The value of this subkey includes the disk drive where the segment was loaded, but it cannot be accessed until after segment loading is completed.

It is strongly recommended that segments use the segment descriptors provided to "self-describe" the segment and allow the `COEInstaller` to perform the installation chores. This ensures a consistent approach for all segment installations, and avoids potential conflicts between different segment installation approaches.

6.7 NT COE Descriptors

The descriptor files defined in Chapter 5 apply to the NT-based COE as well. This section is provided as a quick reference for items that are NT-related. Refer to Chapter 5 for complete discussion of each of the descriptors discussed below.

General comments follow.

- NT segments are required to use `SegInfo` for descriptors; that is, NT segments may not use individual descriptor files since these are obsolete. All obsolete conventions are explicitly invalid for NT segments and are flagged as errors by `VerifySeg`.

- Pathnames must be given using '\' in conformance to the Windows environment.

- Segments should not need to specify a disk drive because such designations are considered to be advisory only. For backwards compatibility, when a disk drive designation is given, it and any associated pathname must be enclosed in double quotes. This is required so that the tools can distinguish between use of ':' as a field delimiter for descriptor lines, or as a separator between a disk drive name and a directory pathname.

- In accordance with commercial standards, executable descriptors shall have either a `.EXE` extension (for compiled programs) or a `.BAT` extension (for batch files). This applies to the "scripts" used in the installation process: `DEINSTALL`, `PostInstall`, `PreInstall`, and `PreMakeInst`. Segment descriptor files may optionally have a `.TXT` extension.

- The `SYSTEM_ROOT` environment variable is set to indicate where the Windows system directory is located. This environment variable may be used in the installation-related "scripts" at install time.

Comments related to specific descriptors follow.

AcctGroup

NT account groups must omit the *shell* parameter. It has no meaning in Windows.

COEServices

The `$GROUPS` and `$PASSWORDS` keywords are not supported for NT platforms. `VerifySeg` generates a warning if a segment descriptor contains these keywords.

DEINSTALL.EXE and DEINSTALL.BAT

Chapter 5 indicates that `DEINSTALL` is executed prior to a segment being removed from the system. A segment that does not include a `DEINSTALL` descriptor is a permanent segment and may be updated, but not removed. In many situations, it is desirable for the segment to be removable, but there are no actions that `DEINSTALL` must perform. For this reason, the NT-based COE allows `DEINSTALL` to exist as a zero-length file and it may exist as a file with no extension.

FileAttribs

Because file permissions are different between the UNIX and NT environments, `FileAttribs` is operating system specific. The COE tool `MakeAttribs`, when run on an NT platform, will create a proper `FileAttribs` file for NT segments. C style `#ifdef` preprocessor statements may be used to combine a UNIX and NT `FileAttribs` descriptor.

Hardware

The *diskname* field for the `$PARTITION` keyword must be a disk drive name. For example, to indicate that a segment requires 20MB on the `F` disk drive, the proper `$PARTITION` statement is

```
$PARTITION:"F:":20480
```

Network

The `Network` descriptor is not presently supported for NT platforms. `VerifySeg` will issue a warning if a `Network` descriptor is found for an NT segment.

Processes

The `$RUN_ONCE` keyword identifies process that should be run the next time the system is started. This keyword requires authorization by the cognizant DOD Chief Engineer because of potential security and performance risks.

Registry

The `Registry` descriptor allows the segment to have the COEInstaller create registry key entries.

ReqrdScripts

Environment extension files are not supported for NT platforms. Therefore, the `ReqrdScripts` descriptor is not supported for NT platforms. `VerifySeg` will print a warning if this descriptor is present.

SegName

The `$COMPANY_NAME` and `$PRODUCT_NAME` keywords allow a COTS segment to specify company and product names for the registry. These are added by the `COEInstaller`, and must not be specified if the COTS product creates registry entries itself.

SharedFile

This descriptor allows the segment to identify shared DLLs.

7. Web-Based Applications

The DII COE includes a collection of COE-component segments to support Web-based applications. This provides a foundation for the development of Web-based segments within the DII COE, and for mission applications built on top of the COE. The Web component segments provide services and infrastructure for the delivery of HTML files[54] from a Web server to a Web browser. One of the key goals in adding Web capabilities to the DII COE is to foster sufficient discipline to prevent anarchy, while permitting a flexible Web runtime environment.

The COE Web component segments are designed to meld diverse system and operator requirements while benefiting from advances in Internet technology and functionality. Evolution of Web component segments is driven by several factors:

[54] The term "HTML file" is used throughout this chapter to refer to hyperlinked pages that may be traversed from a Web browser. These files may be documents or HTML pages in the traditional sense, but may also contain "executables" in the form of applets or other techniques.

- architectural freedom for creativity and rapid progress,
- reduction of site maintenance workload,
- improved configuration control,
- improved service to customers with low-bandwidth,
- customer demand for access to (and sharing of) remote data sources, and
- the rapid pace of Web innovation.

This chapter is devoted to explaining the COE Web component segments and to providing implementation guidance for creating Web mission-application segments. It should be noted that the majority of users will likely use PCs, so this is considered the target client platform for Web development. However, the principles and techniques presented here work equally well for the UNIX environment.

Section 7.1 discusses fundamental COE Web concepts. Section 7.2 describes Web administration and user accounts. Section 7.3 contains miscellaneous information pertinent to developing Web segments, including an overview of HTML requirements for the COE Web. Section 7.4 describes what happens when Web segments are installed, and section 0 completes the chapter with a brief discussion of supported configurations.

7.1 Fundamental COE Web Concepts

All Web-based segments must be DII-compliant. This applies to Web-based COE infrastructure software as well as mission-application software. The principles that govern how segments are loaded, removed, or interact with one another are the same for all DII COE segments, but COE Web component segments are treated more strictly because they are the foundation for a Web-based application.

It is important to recognize that just because a Web segment is part of the COE, it is not necessarily always present or required. Considerable flexibility is offered to customize the environment so that only the segments required to meet a specific mission application need be present at runtime. This approach allows minimization of hardware resources required to support a COE-based system.

7.1.1 COE Web Component Segments

The DII COE provides a collection of component segments to provide the architectural framework for managing and distributing data from a common Web server. Management Services include system administration, security administration, and segment registration. System administration includes the ability to monitor system performance. Security administration includes a tool for managing Web-based access control lists (consistent with the format required by the Web server), and the ability to create and manage Web user accounts.

These services are independent of any particular segment. It is anticipated that diverse segments will be able to coexist, providing access to a wide variety of data sets. However, integration and/or cooperation between segments is the responsibility of the segment developers.

7.1.1.1 Web Servers

A Web server is required to provide the interface between users and Web-based applications. The DII COE provides a Web server as a COE-component segment, thereby eliminating the requirement for individual Web segments to include a Web server. A Web mission-application segment shall *not* include its own Web server. It is required to use the Web-server segment provided by the DII COE. This is in keeping with the overall DII COE philosophy of not duplicating DII COE services.

A site installation may contain multiple platforms set aside to function as Web servers. The platforms may also serve other functions, but it is expected that sites will use firewalls to isolate Web servers from the rest of the world. For this reason, the COE requires that all Web-application segments be loaded on a machine that already contains a Web server. That is, the application interface must be on the Web server but other parts of the system that the application needs to access (e.g., database server) need not reside on the Web server.

7.1.1.2 Web Browsers

The COE includes a Web browser, and COE-based systems will use that browser. However, non-COE based systems can use their native browser to access services provided by the Web server. Web technology is evolving at a rapid pace, so the Web server must accommodate and address evolving Web standards. The DII COE Web server does not restrict or constrain the types of HTML files (Virtual Reality Modeling Language [VRML], executable content, etc.), subject to appropriate security considerations.

7.1.2 Web Mission-Application Segments

Web-application segments shall place their HTML files in the directory

```
$DATA_DIR/local/SegDir/pub
```

where *SegDir* is the segment's assigned directory. The HTML files are thus placed in the local data directory on the machine that hosts the Web Server(s). The COE creates a symbolic link from

```
COE/Comp/WebSvr/data/pub/SegDir
```

to this directory at installation time. The reason this symbolic link is created is so that the Web server can access HTML files provided by the segment. Only Web component segments are allowed to modify HTML files created by other applications, which is typically for the purpose of inserting value-added HTML tags prior to delivery to a browser. The importance of these principles cannot be overemphasized to avoid environmental conflicts between software components.

7.2 Web Account Groups

Operating systems such as UNIX and NT assign individual login accounts for users. There may also be configuration files for login accounts that establish a runtime environment context. The Web environment presents a different set of requirements for user accounts since there is no need for a standard UNIX or NT login account or any of the associated configuration and environmental files. Instead, Web user logins are validated by the Web server that is also responsible for enforcing access control, including restrictions based on the combination of user account and IP (or IP class) on a directory-by-directory basis.

Web account groups can be used to share access privileges among a collection of users according to how they will use the system. This technique is used in the COE to identify three distinct account groups:

- Web System Administrator Accounts,
- Web Security Administrator Accounts,
- Normal Web User Accounts.

Other account groups may exist for specialized system requirements, but all account groups follow the same rules. Within a Web account group, profiles can be created as with normal COE account groups defined in Chapter 2.

7.2.1 Web Security Administrator Account

Security administration in the COE Web is implemented through a special Web account for managing the Web user account database. Precise functionality of security management is dependent on the Web server and its configuration. The role of the Web security administrator includes:

- Ability to create individual Web login accounts
- Ability to create operator Web profiles
- Ability to review the Web server error and user access logs

The Web security administrator need not be the DII security administrator, but this is recommended to centralize security management.

7.2.2 Web System Administrator Account

System administration consists of a specialized collection of functions that allow a system administrator to perform maintenance, monitoring, and configuration operations. The role of the Web system administrator includes:

- Ability to create and to restore backup tapes
- Ability to monitor and configure the Web COE-component segments
- Ability to establish site-specific products and links for user access
- Ability to review the Web server error and user access logs
- Ability to tailor Web applications (consistent with the application design) to balance overall system performance

The Web system administrator need not be the DII system administrator, but this is recommended to centralize system administration.

7.2.3 Web User Accounts

Most operators will not require, nor will Web administrators grant access to, capabilities described in the previous sections. Most system users will be performing mission-specific tasks. The precise features

available depend upon which mission-application segments have been loaded and the profile assigned to the operator.

The COE establishes individual operator login accounts and stores user-specific data items, including profile information describing which options and services are available to the operator. Since users do not directly access Web segments (i.e., the Web server provides the interface between the browser and segments), many of the normal DII COE requirements for additional user-specific directories and services do not apply.

7.3 Miscellaneous

The use of server-side includes (SSIs)[55] is not allowed because of the additional complexity it imposes on the Web COE in the control of data. The subsections that follow provide additional requirements and information for Web segments, beginning with HTML specifications.

7.3.1 HTML Specification

The rapid pace of innovation in Web technology makes it difficult to standardize on the exact HTML syntax that Web-application segments must support. Indeed, any HTML standard is only as good as the browser implementation. HTML version 3.2 is the latest standard, but it is not fully featured. For example, it lacks the <FRAMES> tag. Furthermore, version 3.2 is not fully supported by all popular browsers (e.g., Netscape 3.0 does not support style sheets). DII COE Web segments must, as a minimum, support HTML 3.2 and frames. The application segments should be designed to work with browsers that do not support frames or all parts of the HTML 3.2 specification, or at a minimum notify "disadvantaged" users. The Web server must be able to support HTTP 1.0 and HTTP 1.1 transport protocols.

An HTML file consists of a document head and a document body, as identified by the HTML tags <HEAD> </HEAD> and <BODY> </BODY>. For the purposes of this section, it is convenient to separately discuss the data content within these tags.

7.3.1.1 HTML <HEAD>

The HTML head shall contain three important data elements:

- Title (determined by the Web segment that creates the HTML file)
- Key words (used by Web search engines to identify and index Web sites for global search)
- Expiration date (using EXPIRES) to assist browsers in automatically rejecting out-of-date information

Key words or subjects are appended to META tags and significantly facilitate the ability of Web search engines to locate data services at other Web sites. These tags must not contain classified information (even if the entire system is running on a secure network); access to the underlying data will only be granted to users with valid accounts at the associated Web site. The use of Web search technology (bots, crawlers, spiders, etc.) requires coordination with each Web site since a login/password is required for any DII-compliant Web server connection; importantly, access to data by search engines can be provided for HEAD-only information (once a login and password have been authenticated for the special "HEAD-only" account). Additional restrictions can be implemented using access control lists in each directory. A segment that only generates dynamic, on-the-fly, HTML files may create a static HTML file with identification information specifically for the purpose of identifying the segment's information content. The HTML file shall be called *segment_name*.htm. The format of this HTML file shall be a standard HTML file with META tags for key words and subjects, thereby allowing HEAD-only searches to gather profile information.

7.3.1.2 HTML <BODY>

The DII COE approach is to specify the minimum set of HTML tags that are currently supported, or likely to be supported, by the popular browsers (e.g., from Microsoft and Netscape). The COE does not explicitly

[55] Server-side include is a technical process whereby HTML pages are parsed by the server prior to the page being sent to the client. This allows the server to dynamically insert information into the page before it is sent to the client.

prohibit the use of additional HTML tags as required by a Web segment to satisfy its requirements, but provision may be made by the segment developer to alert "disadvantaged" users to potential problems.

Each Web segment is responsible for properly classifying every HTML page that it creates. The classification marking should be placed at the top and bottom of the HTML page (there is no notion of page breaks in HTML).

7.3.2 User Interface

Innovations to the Web interface offer improved user interaction and navigation via the FRAME tag, Java, JavaScript, and ActiveX functionality. These techniques enhance the user interface capabilities of Web-based applications, but at a price. The security community has expressed concerns about the potential for viruses or other malicious software spread through Java applets and applications. Developers should note that DISA is presently formulating a policy on Java usage for creating applets, and for execution by Java Virtual Machines. An update will be issued when an appropriate policy and guidance have been formulated.

Refer to the *DII User Interface Specification* for further style-related guidance in developing Web-based applications.

7.4 Installing Web Mission-Application Segments

Installation of Web segments, whether they are COE-component segments or mission-application segments, is accomplished as for all other segments. There are some special considerations for Web mission-application segments.

Web mission-application segments must reside on the same platform as a Web Server. The COE installation tools will not allow a Web-application segment to be loaded unless there is a Web-server segment already loaded.

During installation of a Web mission-application segment, two symbolic links for use by the Web server are established, namely

- A link for accessing Web pages from the directory
  `COE/Comp/WebSvr/data/pub/`*`SegDir`*
to the directory
  `$DATA_DIR/local/`*`SegDir`*`/pub`

- A link for accessing Common Gateway Interface (CGI) programs from the directory
  `COE/Comp/WebSvr/data/pub/cgi-bin/`*`SegDir`*
to the directory
  `$DATA_DIR/local/`*`SegDir`*`/cgi-bin`

Also, the `httpd.conf` file will contain an "execution" statement and a "pass" statement of the form:

```
Exec /cgi-bin/* /h/COE/Comp/WebSvr/data/pub/cgi-bin/*
Pass /*     /h/COE/Comp/WebSvr/data/pub/*
```

Here are two examples to clarify the navigation process for locating HTML files and CGI programs. Suppose a segment called `MYSEG` uses a gateway program called `TEST`, which is referenced in an HTML page as

```
FORM ACTION=/cgi-bin/MYSEG/TEST
```

This program will be found by the Web server as follows. First, the "execution" statement is used to convert the file's location to

```
/h/COE/Comp/WebSvr/data/pub/cgi-bin/MYSEG/TEST
```

Then, the symbolic link transfers this reference to

```
$DATA_DIR/local/MYSEG/cgi-bin/TEST
```

As a second example, suppose an HTML page contains a hyperlink to a file

```
HREF=http://hostname:9000/MYSEG/DOC
```

Once the connection is established to a DII-compliant Web server, then the "pass" statement is used to convert the location of the HTML file to

```
/h/COE/Comp/WebSvr/data/pub/MYSEG/DOC
```

Then, the symbolic link transfers this reference to

```
$DATA_DIR/local/MYSEG/pub/DOC
```

Note: The DII COE establishes the SUID for the Web server. Applications must not be created which depend upon a particular setting. Instead, segments shall allow the COE segment installer to handle such details automatically.

All HTML files in `$DATA_DIR/local/SegDir/pub` must be readable by the Web server. The Segment Installer will automatically set the permissions on Web HTML files when the segment is loaded. Furthermore, all HTML files created by the segment for Web access must be placed in `$DATA_DIR/local/SegDir/pub` and must be readable by the Web server.

7.5 Supported Configurations

The COE Web component segments establish an open architecture that is not tied to a specific Web browser. They use industry standards for interfacing to the Web server (e.g., CGI) and de facto standards for HTML (as contained in HTML 3.2 and extended by the leading browsers). The HTML specification has not progressed to the point where a common presentation is guaranteed across all popular browsers.

The list of supported Web servers and Web browsers is heavily dependent on market forces as the Web industry evolves to satisfy commercial requirements. In general, it is desirable to minimize any specific dependencies on a particular browser or server. Presently, there is no commercial agreement on Web server standardization and much work remains to evaluate the leading candidates. Refer to the DISA DII COE Chief Engineer for the current status on server and browser requirements.

Precise hardware requirements in terms of memory, disk space, etc. is a function of many factors and cannot be specified in a general context. Refer to the DISA DII COE Chief Engineer for hardware configuration options.

8. DCE-Based Applications

The DII COE is designed to support applications using the distributed client/server computing model. There are many ways to implement a distributed client/server environment. The DII COE provides the Open Software Foundation's (OSF) DCE as a baseline for distributed architecture/standards. To be DII-compliant, there is no requirement to use DCE as the baseline for a client/server implementation or that segments be client/server-based. However, if the application uses RPCs (Remote Procedure Calls), they must be compatible with DCE RPCs.

DCE is an integrated set of services that supports the development, use, and maintenance of distributed applications. A set of written standards and a package of developer's software are available from the OSF.[56] Based on these, a large number of applications have been written by various software vendors for end users. Use of DCE is not restricted to UNIX environments. Clients or servers may operate on other operating systems, although most applications employ Microsoft Windows or Windows NT clients and UNIX servers.

The purpose of this chapter is to provide the minimum essential information necessary for developers to begin creating DCE mission applications. It is not a tutorial on DCE, nor does it provide an in-depth discussion of development tools, management procedures, or compliance criteria (in the sense of DCE standards). Developers using DCE should refer to OSF or vendor documentation for general guidance on DCE.

The DII COE provides a COTS implementation of a DCE server and a DCE client. Developers shall use these rather than providing their own copy of an alternative COTS DCE product. This is required of all segment developers, including mission-application developers, because the end COE-based system is likely to be installed on a LAN that includes multiple COE-based systems where incompatible DCE products could create interoperability and administration problems.

> **Note:** Segments must specify the DCE attribute to make use of any of the DCE features described here. A fatal error message will be generated by the `VerifySeg` tool if a segment references DCE segment descriptors but fails to indicate that it is a DCE segment.

[56] DISA maintains a facility called the Operational Support Facility in the Washington, DC area. Throughout this chapter, unless otherwise indicated, OSF refers to the Open Software Foundation and *not* to DISA's Operational Support Facility.

Refer to Chapter 5 for information on how to specify the DCE attribute for a segment.

8.1 DCE Overview

OSF's DCE is commercial software that provides a comprehensive set of services that support the development, use, and maintenance of distributed applications. DCE allows diverse systems to work together cooperatively and masks the technical complexities of the network. Because DCE is independent of the operating system and network, it is compatible with many diverse environments.

The strength and appeal of DCE stem from its ability to make a group of loosely connected systems appear as a single system to Information Systems (IS) staff, end-users, system administrators, and application developers. Applications executed under DCE take advantage of untapped resources on networks by finding the platform best suited for a particular job. Similarly, complex tasks can be easily split among multiple computers on the network to reduce computing time and improve performance. From a security perspective, users in a DCE-enabled computing network need only log in once for access to all network platforms.

Many compare the OSF's DCE to wiring or plumbing because it provides the underlying transport layer that enables distributed client/server applications to interoperate across a heterogeneous environment. DCE currently consists of the following services:

- RPCs
- CDS
- Distributed Time Service (DTS)
- DFS
- Security Service
- Threads.

8.1.1 Remote Procedure Call

The key to making many disparate resources function logically as one system within DCE is the RPC. In DCE, RPCs let multiple computers execute applications, or parts of applications, on the platform chosen by the developer as best suited for the task.

The RPC makes a wide variety of application capabilities possible that were previously either impossible or extremely difficult to implement. These capabilities include the following:

1. allowing multiple clients (in a client/server network) to interact with multiple servers, and multiple servers to handle multiple clients simultaneously,

2. the ability for clients, through DCE's Directory Services, to identify and locate network users by logical service name,

3. protocol independence across the network for any platform, and

4. secure communications across the network.

8.1.2 Cell Directory Services

The DCE CDS provides a single naming model throughout a distributed environment. Directory Services let users access network services, such as printers, servers, and other network platforms, by name, without the necessity of knowing where the resource is located within the network. This lets users access a network resource even if the resource has been moved to a different physical network address.

The CDS can make use of its built-in X.500 Global Directory Service (GDS) for locating resources in external cells, or can make use of Domain Name Service (DNS) for this purpose. Cell names are constructed differently depending on which approach is selected.

- The DII COE will use DNS to locate external cells, and therefore will use DNS-style cell names.

8.1.3 Distributed Time Service

DCE DTS allows multiple platforms to work together to share information without timing problems that might affect event scheduling and duration. DTS regulates system clocks on each network computer so that they match each other. Clocks are synchronized, and the service ignores faulty system clocks. The DCE Time Service uses authenticated DCE RPC so that, unlike the Internet Network Time Protocol, the DCE global clock synchronization is secure. Also, to support network sites that wish to use time values from outside sources, DTS supports the Network Time Protocol standard. The DCE Time Service also includes a published Time Provider Interface to allow it to receive inputs from other reliable time sources, such as Global Positioning Satellite (GPS) or other military systems.

- DCE DTS provides intra-cell clock synchronization in the DII COE. Inter-cell synchronization is not supported.

8.1.4 Distributed File Services

The DCE DFS is a fundamental element for information sharing in DCE-enabled networks. It is one of many facilities that could theoretically be built on the foundation provided by DCE's Core Services. DFS unites the file systems of all network nodes for a consistent interface, making global file access as easy as local access. It replicates files and directories on multiple network machines for fast and reliable access, even when communication lines and network hardware fail. It also caches copies of currently used files at the requesting node to minimize network traffic and provide fast data access.

> **Note:** DFS is not presently provided as part of the DII COE. It is described here for completeness sake. Specific communities may implement DFS on top of the DII COE. Information in this chapter about DFS describes it as it is planned to be used by the GCCS community. This may serve as a useful model for other mission domains.

8.1.5 Security

While security maintenance and administration are simplified for one central system behind a glass wall, security for dozens of computers scattered across a wide area network, all operating as a single entity, is much more complicated. DCE's Security Services ensures distributed security. The Security Service software layer is made up of three mechanisms: authentication, authorization, and user registry. DCE invokes these services through the RPC, which maintains the integrity of information passed across the network.

The authorization mechanism grants authorized users access to resources and rejects requests from unauthorized users. DCE implements Access Control Lists (ACL) based on a draft POSIX standard that provides a fine-grained object/operation security authorization model.

The user registry permits users to access multiple network resources through a single password and single login. The registry is a single database of user information that may be replicated around the network. User passwords and security-related attributes are centrally stored and universally available.

Many security features, including auditing, delegation, and a registry extension to support non-UNIX systems, are provided by DCE. Improved security is one of the primary motivations for the movement to DCE for DII applications. OSF DCE provides the following significant features related to security:

1. DCE Authentication provides a secure mechanism (unforgeable) for establishing identity. This prevents a user from compromising the authentication process by using a 'root' account on any machine to project UNIX credentials.

2. Authorization for execution of applications is based on DCE credentials in addition to UNIX credentials. The granularity of execution control on a base UNIX system is limited to an owner/group/world model that is not sufficiently flexible. As a result, almost all applications are set to enable world execute permission.

3. Authorization for operation invocation is based on DCE credentials. Most existing applications either do not have granular access decisions or have implemented their own means of access control. An example of the latter is a database server that may define roles as a means of protecting classes of operations. New applications and those being migrated need this more consistent means of defining, managing, and performing these operations.

4. DCE security allows a client to securely project its identity, including memberships, in other security groups. This allows authorizations to be group-based rather than user-based.

5. Single-login allows all related access decisions to be based on the same distributed identity. Without this capability, users may be required to login to multiple systems or applications, and security administrators must keep multiple identities and security files in synchronization.

6. Execution auditing records DCE and UNIX credentials. This records the identity of anyone running an audited application (see below).

7. Protection against packet insertion/replay, packet interjection, and eavesdropping can be achieved when using DCE RPCs at the appropriate security level or when using the Generic Security Services API (GSSAPI) to protect data transmitted over the network.

> **Note:** For the near term, security for DII distributed applications will be provided by the DCE Security Service, which is based on Kerberos. The OSF and DOD are exploring ways to link DCE security with DOD initiatives such as MISSI. Other security mechanisms may be provided in future versions of the DII COE as the COE migrates from a software-based security solution to a hardware-based solution.

8.1.6 Threads

The underlying Threads Service is used by several DCE services, including the RPC. Threads are programs that use "lightweight" processes to perform many actions concurrently. Threads are particularly useful in allowing server applications to process multiple requests concurrently. DCE Threads are based on the POSIX threads standard. OSF has designed the multi-threading capability of the Threads Service to be easily accessible by programmers wishing to use it in applications. Most commercial applications using threads are written in C, so these DCE services can be accessed through the C programming language. Bindings exist for Ada, as well as other high-level programming languages.

8.1.7 Client/Server Concepts

DCE is specifically designed to manage the distribution of processing across multiple platforms. It is a powerful infrastructure for building client/server architectures. The client/server computing model for DCE introduces a few additional terms.

1. In the DCE context, a *server* is a single executable program that provides services to clients. An example of a server is a DBMS or a map server that provides map images to a calling application. A site can employ multiple servers to create a more available or more balanced service environment. A DII segment can contain multiple servers each performing some related service.

2. A server implements one or more *services*, each of which is offered through an *interface*. Interfaces are well defined, using the DCE Interface Definition Language[57] (IDL), and are the concrete descriptions of a service. Usually, a server implements at least two interfaces. One provides the operational interface for client requests. The other provides a management interface (e.g., for security). Internally, all DCE servers implement other interfaces used for querying, stopping, or reconfiguring the server.

3. An interface provides access to one or more *operations*, each of which corresponds to a specific function or procedure call. For example, a complex math interface could provide separate operations for complex addition, subtraction, multiplication, and division. The operations within an interface should be very closely related.

4. In DCE clients have the option of locating one or more copies of a server through use of the DCE CDS. The client presents a CDS name (or listing) and, optionally, a resource element (object Unique Universal Identifier [UUID]). The CDS name corresponds with the logical service name rather than a machine or hostname. This indirection allows DCE to provide location independence and employ multiple compatible servers for availability or load balancing.

5. Each operator using DCE is identified with a unique DCE *principal*. A DCE principal has a DCE account maintaining its DCE identifier (UUID) along with its UNIX identity (uid, gid). A DCE principal will map uniquely to a UNIX userid.

6. Each DCE server is also identified with a particular principal. For security reasons, server principals should map to UNIX userids that are not allowed to login (i.e., without a login password). These UNIX userids correspond to the concept of a "system account" (like `uucp`).

7. Although it is not necessary for the client and server to be installed on separate machines, one of the primary reasons for constructing client/server applications is to share access to one or more server resources by multiple clients. Since the segment is the smallest installation unit, the client and server portions of an application are usually delivered in separate segments.

[57] The DCE IDL should not be confused with the CORBA IDL. Both are similar in concept, but differ in implementation.

8.2 DII COE DCE Services

The DII COE supplements the COTS DCE product with a number of tools to assist the developer in creating segments that use DCE and in installing and managing DCE at an operational site. Commercial products are preferable, but many of the tools and features required are not available commercially. The tools discussed in this section, and the DCE-related tools described in Appendix C, are specifically designed for the DII COE rules for DCE applications. In addition, development of DCE guidance for the COE highlighted some issues that must be addressed in order to assist in the development of DCE mission-application segments and implementation of DCE in the COE.

8.2.1 Standard Server Installation

The first part of a DCE server installation process must run as root. Installation of the DCE server has been standardized for the COE and is part of the DCE COE-component segment. Installation uses a parameterized `dcecp` script to create an initial CDS entry and principal for the segment, and give it permissions to create the rest of the structure.

8.2.2 Standard Server Initialization

A secure DCE server must make between 7 and 30 DCE calls on initialization to establish configuration and security information and to register its presence to a CDS. The COE provides a standard server initialization routine.

8.2.3 Standard Client Binding

DCE provides an “automatic” binding routine that will find a suitable server and make a connection. However, this does not work for secure connections or the recommended object model. The alternative requires the client to deal with CDS querying, security, and the possibility of missing servers. The COE provides a standard client binding to allow COE clients to make a single call and not have to deal with this level of complexity.

8.2.4 Standard Reference Monitor and ACL Manager

Secure DCE servers must implement a Reference Monitor (RM) routine to verify the client’s credentials against a server’s ACL, and an ACL manager to maintain application ACLs. For the DII COE, a standard RM and ACL manager are provided as a library routine to every server developer so that security decisions are made in a standard, certifiable manner. The OSF provides a boiler-plate RM, which has been parameterized and “segmented” for use by DII applications.

8.2.5 DCE Verification

The `VerifySeg` tool includes verification of DCE application segments. Refer to Chapter 5 for the appropriate segment descriptor entries and to subsection 8.3.4 for a brief synopsis of the required segment descriptors. COE tools verify that a DCE segment has been properly installed and that CDS entries meet the COE guidelines and agree with the entries in the relevant DCE segment descriptor.

8.2.6 Template Application

Creating DCE segments can be difficult because of complexities within DCE itself. To aid segment developers, the COE Developer’s Toolkit contains an example template application. This application serves as a working model and template for developers of other DII COE applications using DCE.

8.3 Runtime Environment

Many of the security-related objects and concepts within the rest of the COE and UNIX have counterparts within DCE, although the DCE object often has more powerful features and attributes. This section states requirements for the development of client/server applications using DCE. The guidance provided shall be followed by all DII applications using DCE, including applications that do not yet fully comply with the DII COE. Failure to comply with this DCE guidance may result in operational conflicts between applications.

This section begins with a description of the directory structure required for DCE segments. The general structure for segments is described in Chapter 5, but it is useful to collect the information into this section as an easy reference for relevant information. Then, the conventions for CDS and DFS for the COE are described. A summary of segment descriptors relevant to DCE are described and the remainder of this section gives specific information on COE conventions for DCE, organized by server and client.

8.3.1 Segment Directory Structure

DII segments are delivered in accordance with a fixed file/directory structure defined in Chapter 5. Some DCE information is also delivered in UNIX files. Other information, such as CDS information, must be delivered as files and built in CDS as part of installation.

Figure 8-1 illustrates the DII COE directory structure for segments. The shaded portions indicate the additional DCE-specific information which is required. Chapter 5 contains information about segment descriptors that are required for all segments, including DCE segments.

The additional information required to describe DCE segments is as follows:

- IDL for all interfaces shall be delivered in files of the form `*interface*.idl` in the segment's `include` directory, where *interface* is the name of the interface.

- DCE installation/deinstallation `dcecp` scripts shall be delivered in files named `dce_install.dcp` and `dce_deinstall.dcp` in the segment `SegDescrip` directory.

- Additional DCE-related configuration information is recorded in the `DCEServerDef` and `DCEClientDef` segment descriptors. See subsection 8.3.4.

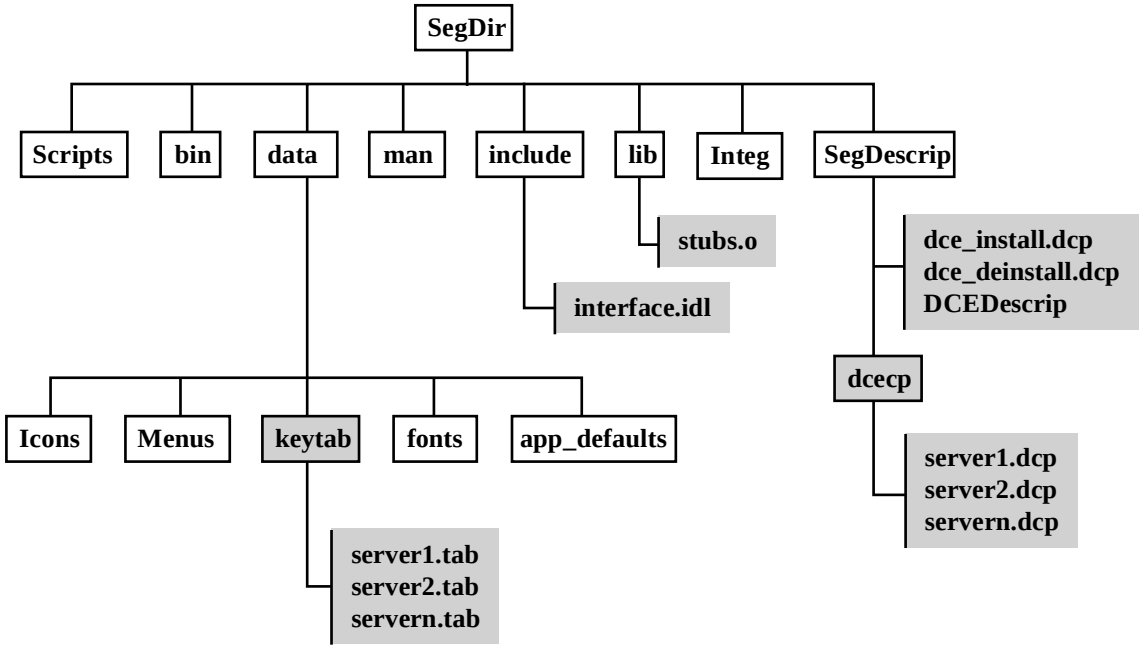


Figure 8-1: COE Directory Structure for DCE Segments

8.3.2 CDS Structure

Figure 8-2 illustrates the CDS structure for a DII COE cell.[58] The following description summarizes the structure:

- Server configuration entries are included under

  `/.:/hosts/`*`hostname`*`/config/srvrconf/`*`servicename`*`.`

These entries will be built by the segment DCE installation script.

- User principal DCE entries have the same name as the UNIX userid. They are included in CDS under `/.:/sec/principal/`*`username`*, but can be referenced in security APIs using just the *username*.

- Server principal DCE entries have the name `hosts/`*`hostname`*`/`*`servicename`*. These entries are referenced in CDS under

  `/.:/sec/principal/hosts/`*`hostname`*`/`*`servicename`*.

[58] Although the CDS directory is described using notation that is similar to the UNIX directory/file system, the CDS is entirely independent from the UNIX file system. The CDS structure includes *containers* that correspond with UNIX directories, and *entries* that correspond to leaf nodes or files.

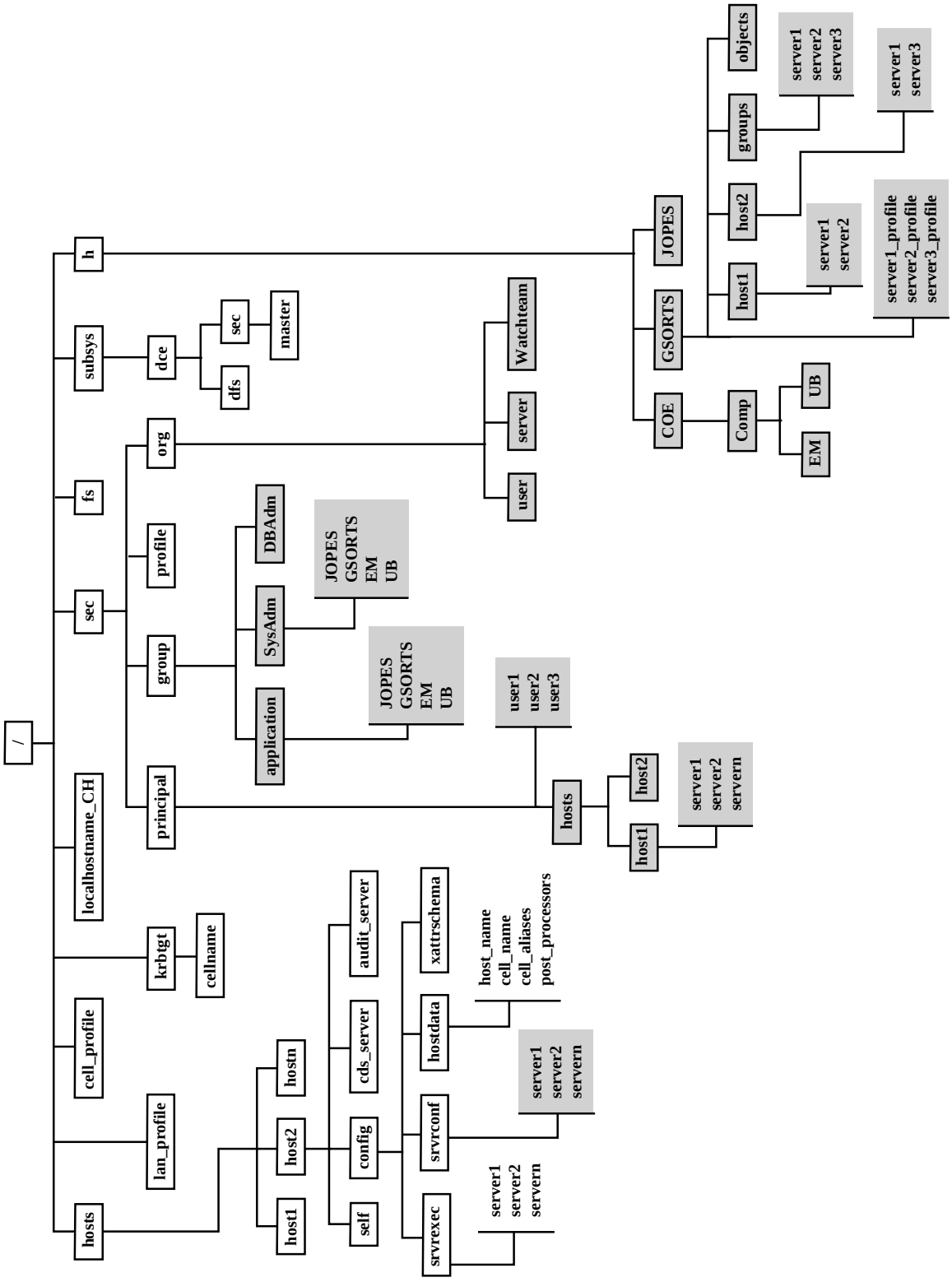


Figure 8-2: CDS Layout for the DII COE

- Security groups and organizations also appear in CDS under `/.:/sec`. (Directories `/.:/sec/group` and `/.:/sec/org` respectively.)

- All server binding entries are contained under `/.:/h`. There is one container for each mission-application segment, named with the segment's assigned directory, and one container for the COE, with sub-containers for each COE segment.

- Each segment container contains a profile entry for each service offered by the segment. This entry is named `/.:/h/`*`SegDir/servicename`*`_profile` and serves as the starting point for all client binding searches.

- There will be a service binding entry for each server for each host on which the server is installed. The entry has the form
  `/.:/h/`*`SegDir/hostname/servicename`*`.`

The name of each entry matches the service name.

- A `groups` container under each segment is used to store any server group entries used in the binding search path.

- An `objects` container under each segment is used to store any object entries used to locate object resources used in binding searches.

8.3.3 Distributed File System

> **Note:** The DFS global cell directory structure is still being designed. COE developers who intend to use the global cell must contact the DII COE Chief Engineer.

8.3.4 DCE-Related Segment Descriptors

Chapter 5 details the segment descriptor information required for DCE segments. A synopsis of the information is presented here as an aid to locating DCE-relevant information. Refer to Chapter 5 for detailed discussion.

- The `$SERVICES` keyword in the `COEServices` descriptor should not be necessary for DCE applications, since endpoints are defined dynamically.

- The `$SERVERS` keyword within the `Network` segment descriptor shall not be used for DCE services. Instead, use the `$DCESERVICE`.

- The segment descriptor `Permissions` may be used, but it is preferable to implement the application using DCE security services.

- The `$DCEBOOT` keyword is provided for DCE servers started by `dced`.

- Include a `$PASSWORDS` keyword in the `COEServices` descriptor to establish a UNIX userid for each server principal.

- Document DFS files used with the `$DFSFiles` keyword.

This information is used to automatically configure, and verify, DCE CDS usage.

8.3.5 Server Issues

This subsection deals with issues involved in the design and implementation of DCE server applications.

8.3.5.1 Naming

The following guidelines apply to the naming of servers, interfaces, CDS names, and operations:

- The service name is the name that represents the logical service provided by a server. In the non-DCE world, this name is put in the `$SERVERS` keyword. The purpose of `$SERVERS` is so that a client does not have to reference the actual hostname of a server. Examples are `masterTrk`, `slaveTrk`, `masterComms`. DCE servers are not tied to a specific host and hence do not use the `$SERVERS` keyword (`Network` segment descriptor). The `$DCESERVICE` keyword is used instead to list the services offered by this segment.

- The following convention shall be used to assign service names: A segment offering a single service shall use names of the form *`SegPrefix`*`_server` where *SegPrefix* is the segment's prefix. Segments offering multiple services shall use *`SegPrefix_service`* where *service* is a meaningful name for the service. This convention will be used in naming many DCE resources associated with a service and will be represented in the text as *servicename*.

- Interface names also will be controlled to avoid duplication. The interface names shall be descriptive of the function of the interface. Each interface shall include the segment prefix. Examples are: `MAP_location`, `MAP_access`, and `MAP_rdaclif` for a segment (whose segment prefix is `MAP`) offering three interfaces. Operation names become the names of remote APIs and shall also begin with the interface prefix or a subset of it (e.g., `location_find`, `access_read`, `access_update`). Operation names shall also be consistent with other COE requirements on naming of APIs.

DCE will automatically provide a management interface for server applications. The only management operation that is controlled is shutdown, which can only be performed by `dced`. If a server wants to restrict other management functions, the server must deliberately disable them using the `dced` management routines: `dced_server_disable_if()` and `dced_server_enable_if()`. Further information on server management can be found in Chapter 8 of the *OSF DCE Application Development Guide--Introduction and Style Guide (Rev 1.1).*

DCE will also automatically add an interface for managing ACLs. The example interface `MAP_rdaclif` mentioned earlier uses the ACL manager API, `rdaclif`. The `rdaclif` interface consists of remote procedures called by `acl_edit` and includes remote procedures to retrieve an ACL, replace an ACL, and test whether a given client is allowed to perform a given operation.

- Names of services and interfaces need not be registered with DISA for approval. Inclusion of the segment prefix ensures that names are unique.

The CDS directory is a naming system somewhat like a filesystem. It uses a similar convention for naming its objects and directories. For example,

```
/.:/h/JOPES/JOPESdb_server
```

Servers typically use CDS for storing information about the location, interface numbers, and objects (i.e., resources) which they offer. Use of CDS naming requires as much rigor as does file system naming.

- Every DCE server segment shall be assigned a directory structure within CDS that parallels its file system location (e.g., `/.:/h/`*`SegDir`* where *SegDir* is the segment's assigned directory). All CDS entries related to this segment are contained within this directory.

In DCE, every DCE server runs under the identity of a DCE principal. Even servers offering the same service but on different machines require a unique DCE identity in order to provide reliable authentication and authorization. DCE principal names are directly tied to the CDS so server principal names can be expressed as a global name or as a name relative to a cell. The global name is considerably longer due to the need to unambiguously specify a principal regardless of the cell from which it originates. Within a cell, the principal can be named without including any cell identifiers because DCE will automatically append the cell information during processing.

- The convention for a DII DCE server is to use the principal name `/.:/hosts/`*`hostname/servicename`*. Each DCE principal contains information relating to a UNIX account that contains its uid. If each principal of the same service had a unique uid, control of server file system resources would be difficult. Each server providing the same service will share a UNIX uid by creating principal aliases. This allows each server to have a unique account with its own password, home directory, etc., yet share the same DCE principal and UNIX account.

- There will also be a security group created for every DCE service. This group will contain all the principals that represent the servers for this service. The purpose of this group is to allow instances of a service on different machines to trust one another. The name for this group will be identical to the *servicename*. Therefore a segment containing multiple services will have multiple security groups. If an application requires additional DCE groups, they will all be prefaced with the segment prefix.

8.3.5.2 Interface Definition

DCE application interfaces are defined using the DCE IDL defined by OSF. All interfaces are identified with a globally unique identifier that ensures that clients bind to a server offering the proper interface. IDL interfaces also allow the identification of versions of an interface. The version numbering scheme allows clients to bind to a server offering any compatible version. Assuming upward compatibility, versioning allows servers to be upgraded independently of clients, and allows old clients to continue to operate with new servers.

- DII-compliant applications shall make use of version numbers and shall provide upward compatibility between versions.

8.3.5.3 Server Registration

Servers record information (bindings) in CDS that identify the interface resources and server location so that DCE clients can find the server when a client requests its service. DCE stores information in CDS structures in three types of records: *profiles*, *groups*, and *server* entries. The record name within CDS that the client accesses can correspond to a specific server, a group of servers, or a CDS profile.[59] Servers within a group are considered to be completely interchangeable, and are selected at random. Profiles allow the selection of alternative servers based on priorities.

Registration of DCE services shall follow the following guidelines:

- The server registration information within CDS shall follow the structure shown in Figure 8-2, which uses the mission-application segment `GSORTS` as an example. Each segment shall have a directory under `/.:/h` corresponding to the UNIX file system directory for the segment (see Figure 8-1). For example, if *SegDir* is the segment's assigned directory, it will have a CDS entry of `/.:/h/`*`SegDir`*. (The segment's assigned directory, *SegDir*, is established when the segment is registered.) Note that

[59] The term *CDS profile* refers to a CDS entry used in locating alternative instances of a service. It has no relationship to the term *profile* used elsewhere in the *I&RTS* to identify applications and resources available to a class of users.

COE-component segments are underneath `/h/COE/Comp` so their corresponding CDS entry is `/.:/h/COE/Comp/`*`SegDir`*. Within the segment directory, individual server instances will be registered under a directory for the host on which the server is installed. The name of the server entry shall be the *servicename*.

- A profile entry shall be created for each service directly under the segment directory using the name *`servicename`*`_profile.` A service can also use RPC groups to collect a set of equivalent servers. Group entries shall be placed under `/.:/h/`*`SegDir`*`/groups`. The segment developer shall use the profile entry as the starting point for binding requests within a client application. This is the name that will be addressed by clients seeking a server.

- The server entry directly under the segment directory will always be a CDS profile entry. The name will have the form *`servicename`*`_profile`. In the simplest case, the profile will contain a single entry, pointing to the server entry for the host on which the server is actually installed. However, by making the client address a profile entry even in this simple case, the server can be moved, or alternative servers implemented, with no changes to the client.

For example, in Figure 8-2, the `GSORTS` segment contains three servers: `server1`, `server2`, and `server3.` The `server1` software is installed on `host1` and `host2`, `server2` is installed only on `host1`, and `server3` is installed only on `host2.` Each server instance is registered in CDS, as shown above, during segment installation. The CDS profile entry `server1_profile` will contain pointers to the two instances of `server1`, with appropriate priorities depending on whether these are equivalent servers or one is a prime and the other a backup. The `server2_profile` and `server3_profile` entries will point to the respective server entries. Note, however, that by simply installing a new instance of `server2` and making the proper entries in CDS, a client will be able to locate alternative instances of `server2` with no application software changes.

- Servers may implement a more complex arrangement of CDS profiles and groups within this structure. A `groups` directory will be created under the application's assigned directory as well as an `objects` directory. The naming of entries underneath `groups` and `objects` is completely under the control of the developer, within the structure above.

The DCE API supports the registration of servers at execution time by the servers. However, to reduce the volume of changes, it is recommended that DII applications build most of the structure in advance, lacking only the specific endpoint information. The specific endpoint (i.e., TCP port) is supplied at runtime to the endpoint mapper and is not stored in CDS. Building the structure in advance also allows it to be constructed using `dcecp` rather than the more complex C-language API. Installation scripts are discussed in more detail below.

- DII-compliant applications shall register servers within CDS during segment installation. The exception to this will be for tactical applications that are installed on systems that are transient members of cells.

  Note: This means that the CDS registration structure is not an indicator that a server exists. The client needs to actually check to make sure the server is alive.

- DII-compliant application servers shall use `rpc_ep_register()` on server startup to register the endpoint with the endpoint mapper. This call is part of `server_intialize()`, as discussed below.

The structure above is designed for the case where service is provided by servers within the local cell. However, DCE has no restriction on the location of the server. A profile entry may point to servers in a foreign cell. This allows a profile to be constructed such that, for example, it would look for a server first in the local cell, then within a near-by cell, and then anywhere. Profiles can also be used to establish

preference for servers based on other criteria as well, such as the performance of the server hardware, or to allow clients to select servers with compatible data representations to reduce data conversion overhead.

The following is required for cross-cell profiles:

- The required approach for accessing cross-cell services is to have a profile in each cell that references local profiles on remote cells. The starting profile has the same name as the profile that is configured into all clients. That is,

  `/.:/h/`*`SegDir`*`/`*`servicename`*`_profile`

  The local profiles will be similar to the profile set up for a single-cell implementation, and will point to all servers within the cell. The primary profile gives priority to servers in the local cell before looking in a foreign cell. This is illustrated in Figure 8-3. The local profile could also be a group if the local servers are equivalent. A profile is required if one server is the master and one is a backup.

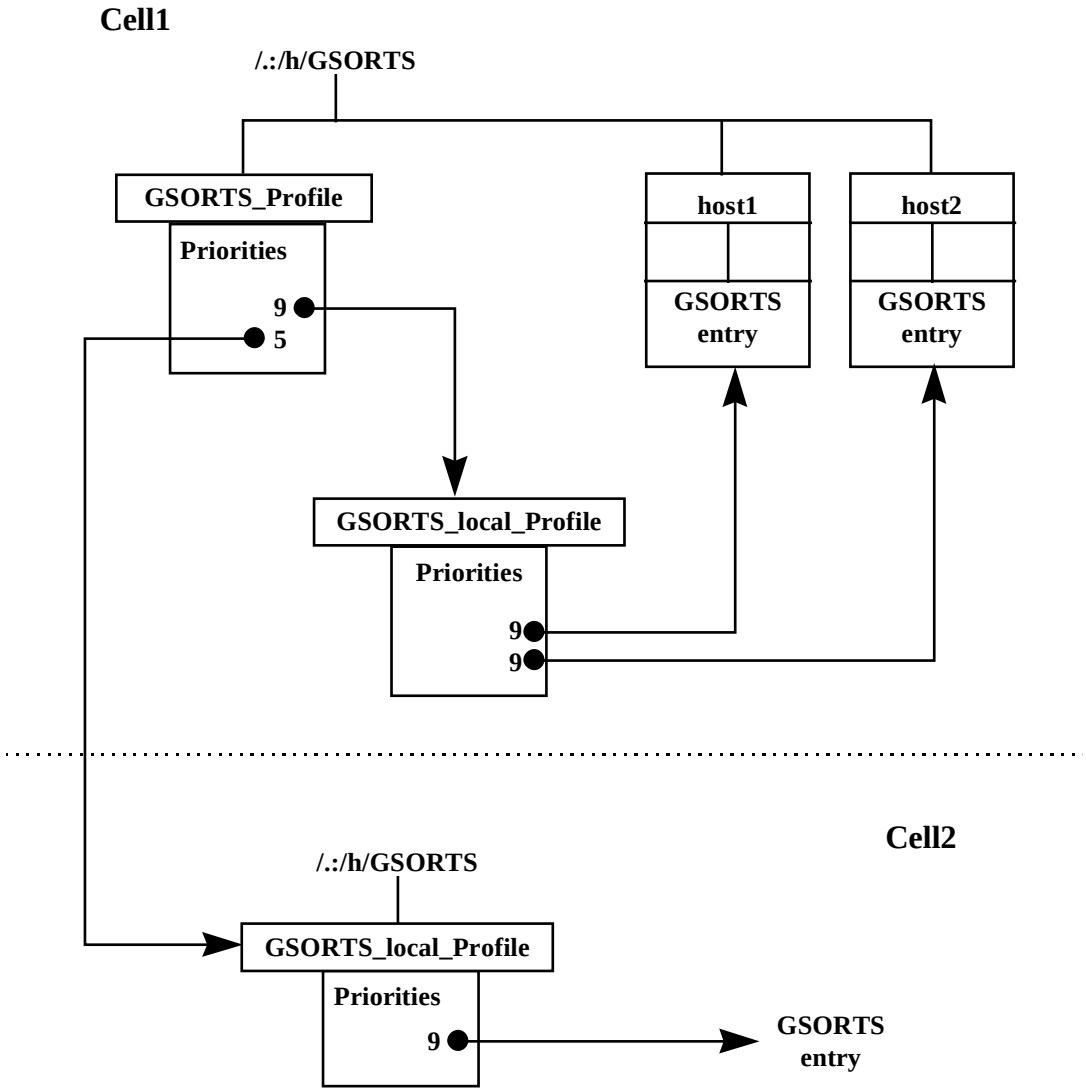


Figure 8-3: Access to Servers in Local and Foreign Cells

8.3.5.4 Server Startup

DCE servers are normally started by means outside of DCE's control after the DCE environment is started. DCE 1.1 introduced a facility for managing the startup and monitoring of DCE servers. This facility is provided by the `dced` daemon and facilitates full security and remote control. When used in conjunction with the client binding recommendations below, servers can be started only as needed, can be restarted in case of failure, and can even be started along with any prerequisite processes as needed. The `dced` process runs as root and is the parent of all DCE servers. Using the configuration information that it stores, it can start the server under any userid/group pair in any directory. The `$DCEBOOT` keyword is used to identify DCE servers started by `dced` at boot time.

The server startup function `dce_server_register()` is provided in order to simplify the development of servers. Unfortunately, not all DCE 1.1 vendors provide this API. The function is included in the `dce_server_initialize()` API discussed below.

8.3.5.5 Configuration

DCE servers contain a large number of configuration attributes that are often hard-coded in the application. The coding of these attributes makes servers hard to change or move and maintain. The `dced` daemon maintains an extensible server configuration database. DCE servers use this database to obtain their configuration information. This database is secure and is remotely manageable. When `dced` starts a server, it establishes an environment for the server based on its configuration record and allows the server to read additional initial information, similar to the windows `.INI` file.

Server configuration information is maintained in CDS under a name of the form

`/.:/hosts/`*`hostname`*`/config/srvrconf/`*`servicename`*`.`

For more information, refer to the *DCE Administration Guide*.

The configuration information which `dced` currently maintains is shown in Table 8-1.

The configuration information is easily extendible by teaching the `dced` about new configuration attributes. Additional attributes can be defined for any DII application as needed. Attributes will be assigned names depending upon their scope. Attributes that are required as part of COE support shall be named:

`/.:/hosts/`*`hostname`*`/config/xattrschema/COE_`*`attributename`*`.`

Attributes that are specific to a server segment shall be named:

`/.:/hosts/`*`hostname`*`/config/xattrschema/`*`SegPref_attributename`*

where *SegPref* is the segment's prefix.

In the case of COE-component segments, adding an attribute requires prior approval of the DII COE Chief Engineer. For mission-application segments, approval is required of the cognizant DOD Chief Engineer.

- Application developers are responsible for creating configuration entries as part of their segment installation scripts (`dce_install.dcp` and `dce_deinstall.dcp` shown in Figure 8-1) invoked at installation time.

Information	Description	
arguments	command-line arguments required by the server	
directory	the home directory in which to start the new server	
gid	the group identity under which the server will run	
keytabs	a list of keytab object UUIDs where the server stores its keys. Although a list is permitted, only the first one is used.	
program	the name of the server program to run	
prerequisites	a list of server configuration object UUIDs which must be running	
principals	a list of server principal names under which the server runs. Although a list is permitted, only the first one is used.	
starton	a list of modifiers for starting conditions (boot, explicit, failure)	
uid	the UNIX uid under which the server will be started	
uuid	a uuid which is assigned to the server object	
services	the DCE information about the operation provided. The following information is defined for each operation:	
	annotation	string describing the service
	binding(s)	protocol sequences which register the service
	flags	modifiers affecting the service's mapping {disabled}
	ifname	the interface name
	interface	the interface UUID
	objects	a list of object UUIDs associated with the service

Table 8-1: `dced` Configuration Information

- If the application is started by `dced`, the DCE daemon will ensure that the appropriate environment (e.g., UNIX uid, gid, home directory, and calling parameters) is established before starting the server. The server will use the `dce_inq_server()` API to obtain its configuration record. There is no requirement for the server to use the configuration information, except to retrieve any relevant extended attribute information and pass it to the initialization routines. Servers not started by `dced` must use the `dced_object_read()` API to obtain this information.

8.3.5.6 Initialization

Every DCE server performs a set of functions in order to initialize. This includes registering one or more groups and entries in CDS (if not already created), and creating and registering endpoints with the endpoint mapper. In addition to these functions, a secure server must establish its identity (login), refresh its login context, and periodically change its password.

- Servers do not normally need to perform CDS registration or unregistration during server startup or cleanup. This is not necessary because the DII COE environment is rigorously defined and because a client does not use the presence of CDS information as indication of server liveliness. Registration is normally performed as part of server installation.

- Servers in a tactical environment may perform registration either at cell configuration time or the first time a server initializes.

Without using a common server initialization API, a server normally performs anywhere from six to thirty API calls. (See the O'Reilly *DCE Security* book for an example of the API calls required for a secure server.) The sequence of calls is nearly identical for all servers in a well-controlled environment like DII because the parameters are defined by the configuration record.

> **Note:** A common `server_initialize()` API is defined and provided as part of the COE to perform these actions. This routine initializes the server, including security, using the server's configuration information.

A server using a special initialization sequence (as defined above) can retrieve its configuration information to perform initialization. Following this guidance will allow servers to be started on demand and can be truly configuration-less.

One of the most critical initialization functions of a server is to register endpoints with the endpoint mapper in `dced`. This too is easily accomplished with the server_initialize API.

8.3.5.7 Security

To write a secure DCE application, besides the application code, the application developer needs to write client code that obtains the proper authentication and forwards it to the server. Clients are usually authenticated by the inherited login context created after `dce_login`. The COE provides a unitary login feature so that DCE login is performed as part of user login. To use authenticated RPC, a client adds a single call to the API `rpc_binding_set_auth_info()`. Clients that use automatic binding will need to use the `binding_callout` option in the ACF file.

Once the client has been authenticated, the server code gets the privileges of the calling client and determines the level of authorization possessed by the calling client. This code is called the reference monitor and it performs the authorization checks. The reference monitor receives the client access request from the server, retrieves the ACL of the object requested and checks the client's authorization against the ACL. The DCE Security Service supports two authorization protocols that can be used with authenticated RPC: DCE authorization and name-based authorization. The DCE authorization protocol is based in part on the POSIX file-protection model, but is extended with ACLs. An ACL is a list of entries that specify a privilege attribute (such as group membership) and the permissions that may be granted to principals who possess that attribute.

- To be DII-compliant, applications shall only use DCE authorization.

8.3.5.7.1 Authentication

Secure servers require DCE security accounts in order to participate in DCE authentication. Each account consists of a principal, and membership in a single primary group and organization. The name of the account is identical to its principal name. DCE security names can be as simple as

```
comms_server
```

or hierarchical such as

```
hosts/hostname/mapserver.
```

- COE hosts shall use DCE principal names that align one for one with UNIX operator names for interactive users. This will allow the use of the integrated login application supplied with DCE. Non-user principals associated with DII servers shall use `hosts/`*`hostname`*`/`*`servicename`*.

The following DCE Security Service application program interfaces can be used to perform login for a non-interactive principal:

```
sec_login_setup_identity()
sec_key_mgmt_get_key()
set_login_validate_identity()
sec_key_mgmt_free_key()
sec_login_certify_identity()
sec_login_set_context()
```

These functions will be performed automatically when using the DCE-provided API, `dce_server_sec_begin()`.

Secure servers must store their passwords in files since they are not capable of normal interactive login. These files are known as *keytab* files.

- For the DII COE, each application segment shall use its own keytab file. Servers shall use names that are of the form *`servicename`*`.tab`. Keytab files will be placed in the directory `/h/`*`SegDir`*`/data/keytab` as shown in Figure 8-1. This directory must have access permissions set so that only the server principal can read or write to it.

Once a server establishes its login context, it is responsible for refreshing the context before it expires and changing passwords before they can expire. The API for managing password expiration is `sec_key_mgmt_manage_key()`. This function does not return and requires a dedicated thread.

The APIs for login refresh are:

```
sec_login_get_expiration()
sec_login_refresh_identity()
sec_key_mgmt_get_key()
sec_login_validate_identity()
sec_key_mgmt_free_key()
sec_login_certify_identity()
```

8.3.5.7.2 Authenticated RPC

A client program calls `rpc_binding_set_auth_info()` to specify how an authenticated RPC connection will be set up. There are three important parameters that must be provided: authentication service, authorization service, and the protection level. Developers should use the following settings for these parameters:

Authentication Service. The default for DCE applications is *dce_private*, which uses private key authentication. No other parameters are valid for DII DCE.

Authorization Service. An application can specify three possible values for the authorization service: *dce*, *name*, and *none*. The value '*dce*' means to pass a Privilege Attribute Certificate (PAC). This is the setting that shall be used for all DII DCE segments.

Protection Level. DCE allows an application to specify just how much the data in an RPC should be protected. These are: *none, connect, call, packet, integrity, privacy*. Integrity provides an authenticated connection between parties and ensures that messages have not been tampered with

in transit. Privacy provides the highest level of protection for the RPC by encrypting the data using Data Encryption Standard (DES). Although the SIPRNET is encrypted using Network Encryption Standard (NES), the DES encrypting provides additional protection from packet snooping within a site.

- DII-compliant applications shall specify at least *integrity*. The *privacy* level should be used for particularly sensitive information.

8.3.5.7.3 Authorization

Once the client has been authenticated, the server must make an authorization decision. The RM is the server code for retrieving the client's PAC. The information from the PAC will be used by the RM to make the authorization decision. While each server can implement its own RM, DCE packages RM code in its library. The intent is for all servers to use this same library code. This will insure that access decisions are made correctly and uniformly.

The ACL is a key part of the Authorization facility. Applications must be capable of establishing and managing ACLs. DCE provides a set of APIs for using ACL managers (`dce_acl_*`).

8.3.5.7.4 Generic Security Service API

DCE provides a method for using DCE security without rewriting applications to use DCE RPC. DCE contains extensions to the *IETF RFC 1508* and *1509* GSSAPI that will allow current applications to use DCE authentication and authorization. GSSAPI DCE extensions can be easily identified since all base GSSAPI entry points start with `gss_` while DCE GSSAPI extensions start with the prefix `gssdce_`. The most important DCE GSSAPI extension is the `gssdce_extract_cred_from_sec_context.` This call returns the Extended PAC (EPAC) which contains the security attributes of the original client and any intermediate servers. The server uses the EPAC to make its authorization decisions. For more information on the DCE Security Service and the GSSAPI, see the following:

1. The Security chapters of the *OSF DCE Application Development Guide-Core Components Volume* and the *OSF DCE Administration Guide-Core Components Volume* (DCE Security Service only).

2. Reference pages (section 3) from the *OSF DCE Application Development Reference*.

3. Reference pages (sections 5 and 8) from the *OSF DCE Command Reference*.

4. Chapter 8, *DCE Security Programming*, Wei Hu, O'Reilly & Associates, 1995.

Note: The DCE Security Service and GSSAPI do not currently make use of Fortezza authentication or encryption. Integration of Fortezza with DCE is under investigation.

8.3.5.8 Auditing

DCE provides an enhanced audit facility consisting of the audit daemon, the `dcecp` control program, and the audit logging client library. An audit daemon exists on every DCE system. Applications audit events by sending RPCs to the audit daemon on the local system. The audit daemons write the audit records to the audit log file, which stores all the event records so that they can be reviewed later. The audit daemon also maintain event filters. Event filters are data structures that determine what events should be logged. Event filters are stored in memory and in files called event selection list (ESL) files. In order to dynamically tailor the audit process, the audit daemon exports an interface that allows the control program, `dcecp`, to change the event filters and expand the range of events that should be audited.

The final process of the audit facility is the audit-logging client library. This allows an application to send audit records to the audit daemon. When an application makes a call to the library, the library checks to see if the event should be audited. If the event filters determine it should not be audited, no RPC is sent to the audit daemon.

This represents a simplistic view of how auditing takes place in DCE. More complex actions are actually taking place including the dynamic updating of event selection lists. The most important point is that applications need only work with the audit-logging API to audit events.

- DII DCE servers shall not write audit information to private audit files. The 'central trail' shall be used to log all audit events.

A complete list of the DCE Audit API routines can be found in the *OSF DCE Application Development Reference, Volume 2.*

An *event* is any action that takes place and is associated with a code point in the application server code. Each event has a symbolic name as well as a 32-bit number assigned to it. Each event number is a tuple made up of a *set-id* and the *event-id*. The *set-id* corresponds to a set of event numbers and is assigned by OSF to an organization. The organization manages the issuance of the event ID numbers to generate an event number. The structure and administration of event numbers can be likened to the structure and administration of IP addresses.

The concept of events allows each DCE implementation to establish audit events for a wide variety of actions that may take place within applications. DCE has established a hierarchy of formats for events. Once again, these are similar to the class structure within the administration of IP addresses. As part of the DCE implementation, DISA will request the assignment of a Format B event number. Format B is designed to be used by intermediate-sized organizations that need the 8 to 16 bits for the event-id. This will provide for the greatest flexibility and growth. Events may also be logically grouped together into an event class. This is a case where it may be more efficient to refer to several events as a single entity/class. Event classes are assigned event class numbers by the OSF. If required, event class number will be requested from the OSF.

8.3.5.9 Threads

DCE automatically implements threads for server applications. The use of threads can be beneficial to allow the server to service multiple clients concurrently. The number of active threads can be controlled by `max_calls_exec` in `rpc_server_listen()`, which can be set to zero if the server software is not "thread safe."

While the use of threads is beneficial and recommended, the following cautions are provided:

- It is well known that threads can conflict with Ada tasking. Use threads with caution with Ada servers.

- Many COTS packages are also not "thread-safe." Calls to databases, windowing systems, and other routines should be done with caution from within a thread.

- Handling of fork/exec and signals is different when threads are used.

When using exceptions with threads, the application must explicitly include the `dce/pthread_exc.h` header file.

8.3.5.10 Installation

In addition to installing software and data to system disk, server installation must also establish entries in DCE CDS as discussed earlier.

- Application segment developers shall include `dcecp` installation/deinstall scripts in the segment descriptor directory. The installation script will build the registration structure in CDS for each interface as part of server installation. The scripts are named `dce_install.dcp` and `dce_deinstall.dcp`. These scripts must contain conditional statements to ensure that some of the entries, such as the *SegDir* container under `/.:/h`, are only created once for each cell. These scripts are executed automatically by the segment installer tool during segment install/removal.

- It is recommended that there be a separate *`servicename`*`.dcp` script for each interface, to simplify configuration and maintenance of server installation procedures. The primary `dce_install.dcp` script must invoke each of the individual service scripts.

- DCE installation is normally performed by the root user logged in using the DCE `cell_admin` identity. In order to reduce the exposure during installation, DCE applications will be installed in a two-step process. During the first step, the minimal set of secure operations is performed. These include:

    1. Creating a DCE account using the principal `segments/`*`SegDir`*.

    2. Creating a CDS directory `/.:/h/`*`SegDir`*.

    3. Setting the ACL for `/.:/h/`*`SegDir`* to permit all functions for the principal `segments/`*`SegDir`*.

    4. Creating a security group `group/segments/`*`SegDir`*.

    5. Setting the ACL for the security directory `hosts/`*`hostname`* to allow the `segments/`*`SegDir`* to create principals below it.

    6. Allowing `segments/`*`SegDir`* to create one account for each service implemented by the segment (`object creation quota`).

    Note: This first installation step is available as a standard utility in the DII COE. It is parameterized based on a set of DCE-related descriptors.

The second phase of DCE installation is performed by the segment-provided scripts (`dce_install.dcp`, etc.) and is run using the account `segments/`*`SegDir`*. It completes the installation process by performing the following for each service:

1. Create a DCE principal (once per cell), usually with the same name as the `hosts/`*`hostname/servicename`* to be used by the server.

2. Create a binding profile for each service of the form

    `/.:/h/`*`SegDir/servicename`*`_profile`

    (once per cell) and add each server entry.

3. Create a server leaf entry (once per instance)
   `/.:/h/SegDir/hostname/servername.`

4. Create server configuration entries (for each instance).

5. Create default ACLs for any server defined objects.

6. Create security entries for the segment under `application` and `group`.

Note: The entire installation process is automated based on information in the segment descriptor files.

8.3.5.11 Server Exceptions

A DCE server must have proper cleanup code. Cleanup code is responsible for graceful shutdown and includes unregistering with the runtime, removing the endpoint from the endpoint mapper, and killing any security management threads.

- Servers wishing to honor a remote 'stop' request, must register an authorization function using `rpc_mgmt_set_authorization_fn()`. This can be used to control other management interfaces.

- Servers shall be prepared to catch signals and perform the necessary shutdown. This can be performed by converting signals to thread cancellation and using a cleanup function (`pthread_cleanup_push`) or using the exception facility to catch the `pthread_cancel_e` condition.

```
comm_status, fault_status op();     /* in ACF file */
error_status_t op ( args ... );     /* in IDL file */
```

  Alternatively, routines can return status by using the return code as follows:

```
op([comm_status, fault_status] st)  /* in ACF file */
```

- All DII-compliant applications shall catch the `SIGHUP` and `SIGTERM` signals and perform a graceful termination. By convention, `SIGHUP` means to terminate as soon as practical, and `SIGTERM` means to terminate immediately.

Note: The initialization API is accompanied by a server termination function so that every programmer does not need to write one.

8.3.5.12 Client-Side Libraries

When a server is being implemented as a reusable service, it is often desirable to develop a client-side library of interface routines to isolate the client from the DCE interfaces. This is the model most often used in commercial packages that provide a callable service. The client deals only with a well-defined call-level interface, independent of the fact that operations are performed by a server. This also allows some library procedures to be performed entirely at the client when there is no need to interact with the server.

- COE services may provide an API library separate from the IDL when that will improve the efficiency or usability of the software. When a library is provided, it shall be delivered in the segment's `lib` directory. Unless authorized by the DII COE Chief Engineer, the library must be provided for all supported COE hardware platforms.

8.3.6 Client Issues

This section provides guidance for client application developers to make use of DCE services to access DCE servers.

8.3.6.1 Binding

Binding is the term DCE uses to refer to a client locating an appropriate server prior to performing an RPC. This is another area where a DCE application writer has plenty of latitude. Binding encompasses issues such as selection of transport protocol, selecting one or multiple servers based on load, location, or other criteria. Ideally, the binding will be resilient and deal with servers dying, stale entries in CDS or endpoint maps, automated remote server startup, and meeting server prerequisites. DCE also supports three methods for binding which affect the way applications are developed (automatic, explicit, implicit).

- It is recommended that applications use the explicit binding method since it is the most flexible. In cases where preserving the API does not permit the use of automatic binding for the client, this does not preclude a server's use of explicit binding. Servers should always use explicit binding so they can obtain client identity and/or client objects.

- One precaution in using explicit binding is that the client is responsible for obtaining another binding should the initial handle fail (i.e. the first server is unavailable). This feature is provided automatically by the runtime when `automatic_binding` is used.

- Automatic binding does not naturally allow for secure binding or for passing an object reference for use in object binding. When using automatic binding, use the `binding_callout` ACF attribute to annotate the binding for security or object purposes. This will register a call-back routine, to be supplied by the client, that can fill in security and object information. Refer to the *OSF DCE Developers Guide - Core Components*.

  Note: The DII COE provides a standard API that clients can use to obtain a binding handle. This simplifies writing client applications and permits the features described above to be implemented as needed.

There are two different binding models available within DCE. In the *service* model, any implementation of a service is assumed to be able to handle any request. This is appropriate for general purpose services such as math routines. The alternative is the *resource* or *object* model, in which servers also identify specific objects for which service is provided. Clients then identify both a service and an object, and DCE will bind to a server that satisfies both requirements. For example, an OPLAN database could identify the OPLANs that it contains, or a map server could identify the maps it can provide. A client could then request "Connect me to a map server that has a map of Bosnia." Different objects could also be used to distinguish between test and "live" versions of a database. The object model can also be used to identify a "role" being supported by a server. For example, the client could request "Connect me to a server that is supporting the 'observer' role." The object model is a little more complex, but provides much greater capability.

- DII COE client/server applications should use the resource model for binding. For the simple case where there is currently no distinction among implementations, each server should register an object corresponding to the server, and the clients should request this object. This establishes the structure for greater flexibility later. It also establishes an object-oriented flavor to interfaces that may ease transition to the use of object request broker technology in the future.

- DII COE client applications need some means of determining the UUIDs of these objects. There are two choices: define the object UUID values in 'header' files, or use CDS as an object catalog. Object entries in CDS shall be placed under the `/.:/h/`*`SegDir`*`/objects` directory or under another

subdirectory under `objects` (i.e., `objects/Maps`). Groups can be used to collect these objects (for example, `groups/Maps` may refer to object entries `objects/Bosnia` and `objects/Iraq`).

8.3.6.2 Exceptions

Exceptions are a means of handling failure conditions which occur during program execution. DCE implements exceptions locally and remotely as a result of an exception occurring during execution on a server. Using exceptions requires the use of a potentially new programming style. DCE uses exceptions internally as a means of conveying the failure status of RPC communications-related failures. The default handling of an exception is a program abort which is not desirable. The choices for an application developer are as follows:

1. Use exceptions by including `dce/pthread_exc.h` and defining `TRY/ENTRY` blocks around code that may raise an exception.

2. Attempt to avoid exceptions by using the `comm_status` and `fault_status` attributes in an ACF file. To this end, new RPC operations should reserve use of the last parameter in each RPC as a means of conveying error status by doing the following:

```
void op ( args..., error_status_t *st);   /* in IDL file */
```

- DII applications shall make provisions for handling exceptions using one or the other of these methods. The latter method is recommended because of its language independence, but either method is acceptable.

8.3.6.3 Security

In DCE, the client is responsible for selecting the security protocol and level, whereas the server maintains the choice of accepting the client's request or rejecting it. The API `rpc_binding_set_auth_info()` is used to specify the client selections. The default protection level is `rpc_c_protect_level_default`. The default authentication service is `rpc_c_authn_default`. The default authorization service is `rpc_c_authz_dce`.

- DII COE clients shall use the DCE authorization protocol along with packet integrity. Applications requiring additional security should justify and identify those requirements appropriately.

In order for a client to initiate a secure transaction with a server, the client must know the server's principal name. This information along with the security level is placed in the binding handle. In the absence of a standard binding interface, the client can obtain the server's principal name using `rpc_mgmt_inq_server_princ_name` or can query the configuration record on the host whose binding was obtained from CDS.

> **Note:** The latter is performed automatically by the COE supplied binding API.

8.3.6.4 Auditing

There is no difference between auditing in a client and in a server. However, auditing is almost always performed in a server rather than in a client. Auditing can be performed by non-DCE applications, but the user or application must perform a DCE login in order to obtain DCE identification information that is inserted in the audit records. See subsection 8.3.5.8 for a discussion of auditing.

8.3.6.5 Threads

While threads are not automatically enabled for DCE clients, the DCE *pthreads* package is available for use by DCE clients. The cautions mentioned under server issues apply to clients. Client application developers should read more about the implications before using threads, particularly with Ada applications. Vendor release notes should be consulted when using threads. Vendors may require the use of special compile flags such as `-D_REENTRANT` or `_THREAD_SAFE_` and may need to be linked with vendor-specific libraries.

8.3.7 Miscellaneous Information and Requirements

This final subsection provides some remaining details for properly using DCE within the context of the DII COE.

- The COE establishes the `CELL` environment variable to contain the current cell name.
- UNIX userids shall agree one-for-one with DCE principals.
- Each UNIX group used with a DCE application shall have a matching DCE group, but not all DCE groups must have a matching UNIX group.
- Account groups do not have a useful analog in DCE, although organizations or groups could fill this function.
- UNIX file permissions are similar to DCE ACLs, although ACLs are much more flexible.

8.4 Distributed File System

DFS offers some unique characteristics as a remote file service product. Some of these capabilities are often replicated by individual applications. Using DFS can provide significant benefits to applications that need to provide coherent file access to a very large community. Using DFS, all sites have access to a single logical file space. In GCCS 3.0 this access is provided by a NFS-to-DFS gateway machine located at each of the GCCS sites. DFS also provides a built-in replication mechanism that can be used to provide rapid file access and high availability. It is fully integrated within DCE and uses secure DCE-RPC as well as DCE's fine-grained access control mechanisms.

> **Note:** This section uses GCCS as an example and the guidance given is specific to the GCCS global cell. However it is also of interest to other DII developers since the techniques applied to GCCS could also be implemented for other areas.

The DFS provides a transparent, secure global file system. DFS has enormous potential for sharing files within and among sites. DFS will be installed within a global cell that has machines at four sites world-wide (DISA, US Transportation Command [TRANSCOM], US European Command [EUCOM], and US Pacific Command [PACOM]). This cell will provide secure, global visibility to current information using automatic replication. All GCCS sites will share files by access to a file server within this cell. Initially, DFS will be used for a limited number of files, but the usage will grow as experience is gained.

DFS provides the following features:

1. *Client-side caching:* DFS is a file service which maintains information about a client and the client's state. Servers are knowledgeable about clients, files in use, and network copies. This allows clients to maintain full disk-based copies of server files to achieve performance rivaling that of local disks. This is accomplished using a token passing scheme. The NFS-to-DFS gateway machines will be configured with large disk caches (dedicated storage) for caching of remote files. The probability of finding cached data within each site, or at least within the theater, will be high and so reduce network-induced delays.

2. *Transparency (POSIX semantics):* DFS supports nearly complete POSIX semantics for file system access. This guarantees consistency of file access to non-replicated files across all DFS clients. For files that are not replicated, DFS will ensure that any file changes are immediately visible to other users of the file. Other systems with stateless implementations have far weaker semantics due to the possibility of having multiple copies in client buffers.

3. *Replication:* DFS divides file systems into smaller hierarchies called *filesets*. DFS can create replicated read-only filesets of a given master writeable copy. Replication provides load balancing and additional availability. A flexible scheme exists for keeping the master and read-only copies in synchronization within selectable time intervals. All reads from the writeable fileset immediately see any changes, while reads from a read-only replica see the change after some delay, usually about 30 minutes depending upon the scheduled replication interval. These consistency controls allow a trade-off between performance and coherence. In general, replication is only used for files that change infrequently.

Note that "immediately visible" is from the perspective of the NFS-to-DFS gateway. Because clients access the gateway using NFS, the NFS consistency semantics apply, and updates may not be immediately seen by the clients.

4. *Backup filesets (cloning):* DFS provides the ability to create a backup of a fileset and to make this backup available online as a read-only copy. The backup is accomplished using an efficient system of file pointers, so that only files changed after the backup take up additional space in the file system. The

use of backup can allow users to recover overwritten or deleted files without administrative help and without doubling file space requirements.

5. *Use of DCE security:* DFS uses DCE security to provide authenticated access and ACLs for granular access. DFS ACLs are based on DCE ACLs, but implement a specific security model that is much more flexible than UNIX file permission bits. ACLs can specify the access privileges afforded to specific users, any local user, users in specific named security groups, users from a specific cell, users from any external cell, any authenticated user, and non-authenticated users.

6. *Initial ACLs:* In addition to specifying ACLs for files and directories, DCE also allows a separate set of "Initial ACLs" to be attached to a directory. These specify the ACLs that will be applied to any new file created within the directory. In addition, "Initial Container ACLs" can be specified to identify the ACLs for any new directories. Among other things, these can be used to allow users to create new files and directories without allowing them to subvert the ACLs on the directory (e.g., granting public access to files in a sensitive directory).

7. *Delegation:* DFS also supports delegation of DCE credentials, which can be used to protect not only who can access a file, but also specify the means of access. For example, ACLs can permit user `john` to access the GEOLOC file through the `GEOLOC` server but prevent `john` from accessing the file without using the server, and can prevent another user from accessing the file even if they use the `GEOLOC` server.

8. *Administration*: DFS supports advanced administrative functions such as hot backup, moving live filesets between machines, quota controls, transactional file system, dynamic re-sizing of file systems and the ability to control groups of files in filesets rather than in file system units.

9. *Location independence/consistency of naming:* All DFS files are accessed by consistent names that do not contain any location information. For GCCS, a file could be in any of the global cell file servers, or replicated in multiple servers. Although GCCS will use a single DFS cell, in general DFS uses CDS to access file systems that can easily span cell boundaries. Every client system has the same file system view regardless of the cell to which they belong.

10. *Wide-area access:* DFS is built on top of DCE RPC that can use TCP, UDP or other protocols. Because of its efficiency, circuits of 56Kbps are adequate to provide wide-area access to DFS servers.

8.4.1 DFS Structure

In general, the DFS file system is a hierarchical structure starting at the `/...` CDS directory. Files in any cell can be addressed just by referencing the DFS filename. The structure of a DFS filename is `/.../`*`cellname`*`/fs/`*`filesystem`*. An example of a system's DFS directory is `/.../gccs.smil.mil/fs/usr/JOPES`. The logical naming of files does not require that the files reside in a specific server. The physical representation may have files in another location or perhaps replicated across several file servers. As a convenience, a symbolic link `/:/` is made to represent the files within the current cell.

> **Note:** In GCCS 3.0, it is anticipated that there will only be a single global cell containing the DFS file space.

One of the primary purposes of DFS is controlled sharing of information. In the C3I environment, information sharing occurs in at least three different dimensions: within an organizational structure (e.g., across a single service or agency); within the unified command structure (e.g., among a CINC, JTF, and supporting commands); and within functional groups (e.g., among operations watchteams at all sites). All of these can be done using DCE security groups. Group ACLs may be attached to any file within a file

structure, but it is most easily understood and administered if the sharing requirements are explicit in the structure. For the GCCS DFS, the file system is organized around these sharing dimensions.

8.4.2 DFS Guidance

DFS should be used for files that meet one of more of the following criteria:

1. Files that are read-mostly (i.e., are read many more times than they are written).
2. Files that require high availability.

- For files that change frequently, there is a tradeoff between currency and the overhead of replication. Changes to non-replicated files are visible immediately, while changes to a replicated file may not be visible for a period of time. The replication update rate can be set by fileset, but a long interval between replication can increase the chances of accessing a stale copy.

- Files that are site-specific must be placed in site-specific directories in DFS. Be cautious when mapping an application data directory into a shared data directory if the application has any hard-coded file names. It is possible for one site to write the file and unintentionally change the values for all sites.

- For GCCS, DFS files will initially be mapped into the local NFS file system on `/GCCS`. All client machines will mount `/...` from the NFS-to-DFS gateway machine. `/GCCS` will be a symbolic link to `/.../gccs.smil/fs`.

- If application-specific directories are used in DFS, the segment installation procedures shall create the directories. Note that the full directory names are site-specific.

- Use symbolic links to map DFS files or directories into the proper place in the local file system. All mapping shall be done at a directory level. System developers are also responsible for constructing symbolic links from the local file system to the global DFS in their installation procedures.

- Do not create a symbolic link from `/.:/gccs.smil.mil/fs/` to `/:/`, and do not use the notation `/:/` within DFS references. This notation refers to the DFS within the current cell. Since all GCCS applications operate outside the global cell, this would create an ambiguous reference if the site implements DFS internally in the future.

- Do not place RDBMS databases into DFS. The DFS file consistency and caching methods do not support the level of sharing required by an RDBMS. It is possible to back up databases into DFS for re-loading somewhere else.

- GCCS application servers, or even clients, may become DFS clients and access the global cell directly. Bypassing the NFS-to-DFS gateway may result in better performance due to local caching and better consistency semantics through avoiding NFS.

8.4.3 Potential Uses for DFS

Global DFS cells can be used in a variety of ways to assist operators and developers, including the following:

1. *Data distribution:* Many sites are using `ftp` as a means of obtaining remote files. The transparency of NFS or DFS is much more powerful than `ftp`. NFS is not well suited for wide-area access and has serious security issues when used across sites. The originator can simply write the data into DFS using any software, and the user can immediately read it using the appropriate application. If the originator changes the file, the other users can almost immediately see the change.

2. *Reference files:* Applications frequently use reference files for maintaining information such as maps, inventory, or flat-file databases. These files are updated by a few sites and are made available to other sites using primitive distribution techniques. DFS also has the ability to use 'cloning' whereby a virtual copy of a file is kept, but with a fraction of the storage costs. Using this feature, the global file system could make available old and new copies trivially.

3. *Secure files:* Files containing security sensitive information should not be kept in NFS file systems. DFS is a secure alternative to NFS. Using DFS, files can be distributed and controlled at whatever degree is necessary.

4. *Mobile Personnel:* Operators who travel regularly to remote sites are probably using non-secure means (i.e., `telnet`) to access files such as e-mail, data files (phone lists) or documents. Both `telnet` and `ftp` can provide access control, but in both cases the user's password is sent unencrypted across the network. DCE provides more flexible security and the password is never exposed on the network. By storing these files in DFS, they can be securely accessed remotely.

5. *DCE configuration information:* Information about site configuration such as its DCE configuration can easily be stored in DFS. Cell backups (critical DCE databases and configuration files) can be done remotely by writing into a global file system.

8.5 Migration Recommendations

Applications must be programmed to use DCE before the application can fully benefit from the power of DCE. It is assumed that the movement to DCE among applications will be gradual. Although not all applications will be re-engineered to use DCE RPCs immediately, they can still take advantage of other DCE services using techniques described in this section.

The next subsections describe four scenarios and identify ways in which DCE services can be used in each case. The example cases are not mutually exclusive in that an application may take advantage of several of them. The first two cases are specifically targeted at legacy applications, while the last two may be used by legacy or newly developed distributed applications.

8.5.1 Case1: Application Startup

A typical application startup scenario in the DII starts with the client workstation displaying a user desktop. The user selects an icon or menu entry, which causes a "button script" to be executed to start a DII application. The application may be local or remote. The desktop ensures that the user is authorized to select the icon or menu item. In the case of an application on a remote application server, the script uses a UNIX command such as `rsh` or `rexec` to start the remote server. The server application then opens a window on the client workstation and begins a dialog with the user.

The `rsh` command requires a level of mutual trust between the application server and the client. It is possible for malicious clients to masquerade as authorized users and run applications for which they are not authorized. This is particularly a problem for legacy applications that run under a distinguished uid, such as JOPES (i.e., not the user's id). Use of a simple DCE wrapper can ensure the user is authorized using strong DCE protection.

Through the use of a transparent DCE *wrapper*, the startup of DII applications can be fully protected using strong DCE authentication and access controls. Instead of invoking a user application, a button-script will invoke the wrapper and pass the name of the user application and any parameters. The wrapper will verify that the user is authorized to use the application, then launch the application. The application receives control just as if the script had launched it directly, so no application changes are required. In addition to performing authentication, the wrapper can audit execution of applications.

The wrapper can be used to launch applications on the client machine or on a remote machine. In the case of a remote application, the wrapper will operate much like the UNIX `rexec` or `rsh`, but will use authenticated DCE RPC to communicate to a remote wrapper server and will use the DCE ACL model. The remote wrapper will authenticate the user, verify that the user is authorized, then set up the application environment before launching the application. Unlike `rexec` or `rsh`, the button script does not need to specify the machine that contains the application. By proper use of the CDS binding information, the wrapper can make a request such as "connect me to a wrapper server on a machine that has the JOPES application."

The wrapper approach has the advantage of allowing full security over execution of DII applications without having to make changes to any applications.

- This temporary approach is permissible only as an interim step for legacy applications as they migrate to DCE. New distributed applications shall be designed as two and three-tier client/server applications making use of RPC. New COE-component segments shall not use this approach without prior approval of the DII COE Chief Engineer. Mission-application developers shall not use this approach without prior approval from the cognizant DOD Chief Engineer.

8.5.2 Case 2: Socket/ONC RPC

Some applications are distributed and use sockets or unsecured ONC RPC to exchange control and data. Some socket applications perform highly sensitive operations, but essentially accept any request presented to the designated endpoint. Even without converting to full DCE RPC, these applications can make use of strong DCE authentication and access control. Socket-based communication is also susceptible to packet insertion attacks.

Existing applications that use sockets or RPC and desire greater security should seriously consider migrating to use of DCE RPC. In many cases the effort to convert to authenticated DCE RPC is not great. However, even if only limited application changes can be made, the use of DCE security is possible using the new GSSAPI. With the GSSAPI, the client application obtains a user credential, which is passed to the server application. The server verifies the user credential through another call to the GSSAPI.

The simplest use of the GSSAPI will get the credential once and pass it only in the first message. This provides some measure of security, but not as much as passing the credential in every interchange. However the latter requires more widespread changes to the application. It also requires the application to periodically refresh the credential before it expires.

The following sequence of calls illustrates the use of GSSAPI:

1. Client calls `gss_init_sec_context` to obtain a security token to pass to the server.

2. Client passes token to the server across the revised socket or RPC.

3. Server receives token and calls `gss_accept_sec_context` to decode the token, then gets a copy of the session key.

If the credential is valid, the server can convert the token (session key) to a DCE client/server, which is used as the subject in the access control decision; otherwise, it rejects the request. The use of GSSAPI is discussed further in subsection 8.3.5.7, Security.

- This temporary approach is permissible only as an interim step for legacy applications as they migrate to DCE. New COE-component segments shall not use this approach without prior approval of the DII COE Chief Engineer. Mission-application developers shall not use this approach without prior approval from the cognizant DOD Chief Engineer.

8.5.3 Case 3: Distributed Databases

Perhaps the greatest potential use of distributed computing in the DII is for distributed databases, using products such as Oracle SQL*NET. This provides some security, but requires duplicate identification of people and resources, increasing administration. It is possible to integrate database security and remote access control with DCE security using COTS.

At least two COTS alternatives have potential for providing DCE security to remote database connections currently using Oracle SQL*NET. The first is to use the SQL*NET DCE product as provided by Oracle. This product provides an Oracle integration of CDS and Security into existing applications and servers. The Oracle database uses the client's DCE credentials for access decisions, alleviating the need for a separate Oracle login. The product also maps DCE groups to database roles, unifying another aspect of security. The ability to map a DCE security group membership into an Oracle role will not be available until the next release. Database servers register in CDS and clients use CDS to locate a database server. Unfortunately, this product is not currently available for all COE platforms.

A second approach is to use Open Horizon's *Connection* product as a means of integrating existing Oracle database clients and servers. It uses essentially the same approach as SQL*NET DCE, and product

availability is immediate. It supports applications using OCI. In addition, this product supports the *de facto* standard Open Database Connectivity (ODBC) remote database connection protocol, allowing access to a large number of other databases and products. Its major disadvantage is that it cannot provide DCE group to Oracle role mapping. It requires that privileged database access be granted to the *Connection* server. It cannot currently be used with applications that use ProC or ProAda embedded SQL, since these use undocumented interfaces, instead of standard OCI.

Note: There are no facilities to directly support either approach in the DII COE. Tools such as *Connection* are under consideration for later COE releases. Developers may make use of these tools with the COE if required. This subsection is provided only to describe a potential migration approach.

8.5.4 Case 4: Distributed Files

Perhaps the easiest way to use the security features of DCE is through use of DFS. For example, the GCCS Global DFS will allow the use of DCE access control, authentication, replication, and consistency controls, with little or no application impact. It reduces requirements for user-initiated FTP and polling.

DFS offers some unique characteristics as a remote file service product. Some of these capabilities are often replicated by individual applications. Using DFS would be a significant benefit to applications that need to provide coherent file access to a very large community. DFS also provides a built-in replication mechanism that can be used for software distribution. It is fully integrated within DCE and uses secure DCE-RPC as well as DCE's fine-grained access control mechanisms. GCCS will use DFS to allow all GCCS sites to have access to a single logical file space. In later versions of GCCS, this access will be provided by an NFS-to-DFS gateway machine located in each of the theaters.

The DFS provides a transparent, secure global file system. DFS has enormous potential for sharing files within and between sites. DFS will be installed to support GCCS within a global cell that has machines at four sites world-wide (DISA, TRANSCOM, EUCOM, and PACOM). This cell will provide secure, global visibility to current information using automatic replication. All GCCS sites will share files by access to a file server within this cell. Initially, DFS will be used for a limited number of files, but the usage will grow as experience is gained.

- Developers planning to use DFS or anticipating a need for DFS for COE-component segments shall contact the DII COE Chief Engineer for more detailed information and guidance. Mission-application developers shall contact the cognizant DOD Chief Engineer to ensure that such usage does not interfere with the COE, or with other COE-based systems.

This page is intentionally blank.

9. Development Environment

The DII COE imposes very few requirements on the process or tools developers use to design and implement software. The COE concentrates on the end product and how it will integrate in with the overall system. This approach provides the flexibility to allow developers to conform to their internal development process requirements. However, developers are expected to use good software engineering practices and development tools to ensure robust products. The purpose of this chapter is to suggest certain development practices that will reduce integration problems, and the impact of one segment on another.

Developers may select compilers, debuggers, linkers, editors, Computer-Aided Systems Engineering (CASE) tools, etc. that are most suitable for their development environment. The compilers and linkers selected must be compatible with the products supplied by the hardware vendors and must not require any special products for other developers to acquire in order to use the segments produced.

9.1 Coding Conventions

This section describes required coding standards for segments submitted to DISA, whether they are COE-component segments or mission-application segments that are part of a DISA COE-based system. These standards are not intended to restrict software development, and for that reason the requirements given are brief.

There are two important points to keep in mind with respect to this chapter. First, the DII COE states requirements for the purpose of ensuring and preserving the integrity of the runtime environment. Therefore, the DII COE is mostly concerned with executables that are produced and not the process used to create them. The COE relies upon other standards (e.g., MIL-STD 2167A, MIL-STD 495, ISO 9000) and practices levied by the cognizant DOD program managers to ensure good programming practices and a quality product. However, certain standards are required because some of the segments produced contain APIs that developers will use to build other segments upon.

Secondly, the DII COE is neutral with respect to programming languages and does not stipulate what programming language to use to write segments. Such decisions are the prerogative of the cognizant DOD program manager. The COE must support segments written in Ada, in support of DOD policy, and C, because of the use of COTS products, and therefore both are addressed in this chapter. Any statements in this chapter, or elsewhere in the *I&RTS*, which appear to state a preference for one language over another are unintentional.

Because most developers are using either C/C++ or Ada, COE-component segments that provide APIs shall be written in either C/C++ or Ada. Availability of APIs for both C and Ada is highly desirable, but will be driven by service and agency requirements. Consult the DII COE Chief Engineer for availability of multi-language APIs, for requirements to produce multi-language APIs for a particular segment, or for support for languages other than C/C++ and Ada.

9.1.1 Language-Independent Conventions

The following suggestions and requirements are language independent.

- Code delivered to DISA shall *not* be compiled with debug options enabled. If available, a utility such as the UNIX `strip` command shall be run on executables to minimize the disk space required.

- Segments should use shared libraries where practical to reduce runtime memory requirements. Segments with public APIs implemented as shared libraries shall also be delivered as static libraries to make debugging easier for developers who need to use the APIs.

- Developers may use GUI tools to build interfaces, but developer's should select tools that are portable across platforms. Segments built with such tools shall use resource files for window behavior rather than embedded code, and must not require any runtime licenses unless approved by the DII COE Chief Engineer for COE segments or by the cognizant DOD program manager for application segments.

- Developers should run all modules through a tool such as `lint` to detect potential coding errors prior to compiling.

- Developers should run all modules through commercially available tools to detect as many runtime errors as possible (e.g., "memory leaks").

- Developers should periodically profile segments by using tools that do a runtime analysis of module performance (% CPU utilization, number of times a function is invoked, amount of time spent in a function, LAN loading analysis, etc.).

- Developers should create a test suite for automatically exercising the segment, especially inter-segment interfaces and APIs, and periodically run the tests to perform regression testing. A formal test plan should be created and submitted with the segment.

- Segments with public APIs shall be delivered with a test suite that covers all public APIs provided by the segment.

- Developers should use a tool such as `imake` for generating `makefiles` that are as portable as possible. If available, the POSIX.2 `make` utility should be used.

- Developers should use automated tools such as `CVS`, `RCS`, or other commercially available products to perform configuration management tasks. Segment developers are responsible for configuration control of their own products. The *I&RTS* does not prescribe a CM plan, but assumes the developer has one as part of good programming practices.

- Developers should periodically rebuild segments from scratch to ensure that all pieces, including data files, are under proper configuration management control.

- Developers should track problem reports in an automated database. This will simplify reporting known problems when the segment is submitted to the cognizant DOD SSA.

- Developers shall separate COTS products from mission-application software because the COTS software may already be available in the DII COE inventory.

9.1.2 Ada

Ada generally requires stipulating fewer requirements than other languages because the syntax and semantics of the language are designed to enforce good programming practices at the compiler level. For example, Ada enforces strong typing so that many common coding errors are caught at compile time.

Ada bindings in particular pose specific areas of concern.

- Developers should design software so that routines that require binding to other languages are isolated into a small number of easily separated modules. This will make maintenance of Ada bindings easier, and make it easier to identify segments that require long-term support for Ada bindings.

- Developers who create Ada bindings to other segments or COTS products within the COE should submit them with their segment so that other developers may reuse them.

- Developers who require Ada bindings to COTS products within the COE (e.g., Motif, DCE) should use commercially available bindings whenever they exist, and whenever it is economically feasible to do so. Products are available which largely automate the process of creating Ada bindings from C header files.

- Developers shall separate submission of their segment and any bindings they create. The segment will be delivered to operational sites while the bindings will be distributed only to other developers.

- Developers should use Ada95 as the language of choice over earlier versions of Ada.

9.1.3 C/C++

This subsection contains requirements and suggestions that are specific to programming in C or C++.

- Developers should use American National Standards Institute (ANSI) C instead of Kernighan and Ritchie C because of the strong typing capabilities of ANSI C.

- Segments that have public APIs written in C shall support ANSI C function prototypes.

- Segments that have public APIs shall support linking with C++ modules. This is done by bracketing function definitions with

```
#ifdef __cplusplus
extern "C" {
#endif

      function prototypes

#ifdef _cplusplus
}
#endif
```

- Segments written in C that have public APIs shall handle the condition where a header file is included twice. This is accomplished by bracketing the header file with `#ifndef` and `#endif` as follows:

```
#ifndef MYHEADER
#define MYHEADER

header file declarations

#endif
```

9.2 Development Directory Structure

Developers may use whatever directory structure is most appropriate for their development process. The installation tools will enforce the logical structure presented in Chapter 5. However, the COE development tools allow segments under development to be located arbitrarily on the disk. For example,

```
VerifySeg -p /home5/test/dev MySeg
```

indicates that the segment to be validated, `MySeg`, is located in the directory `/home5/test/dev`. Similarly,

```
TestInstall -p /home5/test/dev MySeg
```

allows the segment to be temporarily installed from this directory for testing and debugging.

Figure 9-1 shows an example segment directory structure. It has the advantage that it separates public and private code into different subdirectories. `MySeg/lib` contains public libraries provided by the segment, while `MySeg/include` contains public header (C/C++) or package definition (Ada) files The `src/PrivLib` subdirectory should contain library modules that are private to the segment. Similarly, the subdirectory `src/PrivInclude` contains interface files that are private to the segment.

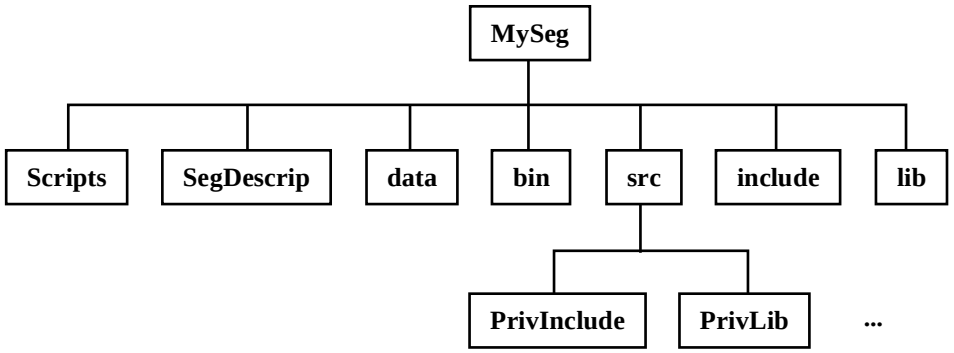


Figure 9-1: Example Development Directory Structure

This directory structure is not mandatory, except when source code is delivered to DISA; otherwise, it represents only one recommended approach. When source code is delivered to DISA, is shall be in the `src`, `include`, and `lib` directories as appropriate.

An advantage of structuring directories as shown in Figure 9-1 is that delivering software to other developers means that only one directory must be deleted: the `src` directory. Delivering the software to an operational site means that only three directories need to be deleted: `include`, `lib` (unless shared libraries are being used), and `src`. It is a simple matter to create automated scripts that can generate tapes for both types of deliveries. An additional benefit is that public and private files are separated in the directory structure for easier management and distribution.

9.3 Separating Out the Development Environment

The COE requires that a strict separation be maintained between the runtime environment and the development environment. This is true regardless of the target platform operating system (e.g., NT, UNIX). For the NT[60] world, most development tools are structured in such a way that the development environment is self-contained in an integrated environment that is accessible from a GUI. For example, both Microsoft and Borland provide an integrated development environment for C++ that provides icon and menu access to compilers, linkers, editors, and other development tools. Both products provide a "directory browser" for identifying the location of source code and libraries, and the target directory for object code and executables. Moreover, they provide an interface for defining parameters such as compiler flags and preserve the settings and all other build-related information in a "project file."

For UNIX, however, integrated development environments are less common place. The next subsection describes an approach for preserving the separation of development and runtime UNIX environments through the use of scripts. The concept is to put all runtime information into one script, and all development information in a separate script. While the approach between NT and UNIX is considerably different, the COE stipulates a fundamental requirement to preserve a separation between the runtime and development environment. Developers shall preserve this separation regardless of the target operating system environment.

9.3.1 UNIX Development Scripts

In the UNIX environment, it is often convenient to locate development scripts in the same subdirectory as the runtime scripts (e.g., subdirectory `Scripts`). The recommended convention is to name development scripts with a `.dev` extension to distinguish them from runtime environment scripts. The `.runtime` extension can *not* be used since this has a special meaning within the COE as explained in Chapter 5.

Developers may define environment variables for locating source code directories, compilers, tools, and libraries. In addition, aliases can be defined as shortcuts for frequently executed commands. None of these examples are allowed in the runtime environment and hence must be placed in a development script such as `.cshrc.dev`.

9.3.2 NT[61] and UNIX Recommendations

The following suggestions are made:

- Define environment variables relative to *`segprefix`*`_HOME` where *segprefix* is the segment prefix. This allows segments to be easily relocated on the disk. (This suggestion is applicable to both UNIX and NT.)

- Use environment variables to define where to place libraries and executables. (UNIX only. For NT, use facilities provided by the development tools for locating libraries and executables.)

- Extend the path environment variable through concatenation - that is

```
set path = ($path $TOOLS)
```

[60] The DII COE for NT is presently available only on PC platforms. Comments in this chapter should be understood in the context of Windows NT for PC-based platforms, even though the NT operating system is available on other commercial platforms. DII COE support for non-PC platforms is dependent upon requirements from the DII COE community.
[61] ibid.

where $TOOLS is the location of the COE development tools (e.g., /h/TOOLS). (UNIX only. For NT, use facilities provided by the development tools for locating tools.)

- Use the same script for all supported platforms through use of the environment variables MACHINE_CPU and MACHINE_OS. (UNIX only. For NT, use facilities provided by the development tools for creating project files that allow multi-platform development support.)

9.3.3 Test Account Group

COE-component segment developers typically create servers that will be used by other segments in the operational system. However, the developers and the SSA need to be able to test the COE-component segments when there may not be available any mission-application segments, or even an account group segment, that will launch the servers and exercise the API interfaces.

To aid the SSA and other segment developers, it is recommended that COE-component segment developers create and deliver with the segment the following:

- **A test account group segment**. This segment should establish the environment that the COE segment is expected to run within and contain details for how to correctly launch the services. This provides a way for the SSA to test the delivered segments and it provides system engineers and designers an example of how the segment was intended to be used.

- **A "Run" script**. Chapter 5 indicates that account group segments must contain an executable that will launch the application. The test segment should also contain such an executable. This encapsulates in one place the information required to properly establish the runtime environment to launch the server and it also identifies the sequence and command-line parameters, if any, required to launch the services.

- **Documentation**. The test segment and "Run" script should be documented to assist the system integrator, potential system designers, and the SSA.

The test segment and "Run" script should be packaged and delivered separately from the actual COE-component segment. This will ensure that the test segment does not inadvertently get delivered to an operational site, or get confused with account group segments that are intended to be part of the end system.

9.4 Private and Public Files

The software engineering principles of data abstraction and data hiding are important in designing segments. *Data abstraction* refers to the process of abstracting structures so that subscriber segments need not know low-level details of how data is physically organized. *Data hiding* refers to hiding data elements that subscriber segments do not need or are not authorized to access directly. Proper implementation of these two design principles prevents segments from affecting each other through inadvertent side effects and isolates one segment from changes in another.

It is also important to hide low-level functions and only provide access to segment functionality through a carefully controlled interface, the API. It is neither feasible nor desirable to make all functions in a segment available due to the sheer number of functions involved and because changing a function that is being used directly by another developer may have significant impact.

These concepts are implemented in Ada through the *package* construct. C, however, does not contain an equivalent capability. The closest approximation in C is the *static* directive that makes a function visible only within the scope of the file containing the function definition. To compensate for structural inadequacies in C, developers must segregate software into public and private files, and into public and private directories. Since header files (e.g., `.h` files) are used to define the interface to C functions, the concept is that header files should be segregated into public and private files while public and private directories are used to provide the same concept for libraries. Moreover, segregation into distinct directories makes it easier to enforce the separation.

9.5 Developer's Toolkit

The Developer's Toolkit contains the components necessary for creating segments that use COE components. The toolkit does not need to be in segment format (it is not installed at operational sites), but it is a set of files and directories that may be downloaded electronically from the online library. Developer's may also contact the DII COE Configuration Management Department to receive the toolkit on magnetic media in relative "tar" format.

The Developer's Toolkit is distributed separately from the target COE-based system. However, components from the operational system (COE-component segments, shared libraries, etc.) are required for development. These may be obtained electronically from the online library, or on magnetic media from the DII COE Configuration Management Department. Classified or very large components will be distributed to developers via magnetic media. The toolkit does not duplicate any components available in the runtime system because this would create configuration management problems in ensuring that developers do not receive two different versions of the same module.

As distributed, the toolkit contains the following:

- API libraries and object code
- C header files for public APIs written in C
- Ada package definitions for APIs written in Ada
- Ada bindings for selected APIs
- API documentation in HTML format[62]
- API documentation in UNIX man page format
- COE development tools (see Appendix C)
- Conventions for creating APIs

The toolkit does *not* contain any products that require a license (compilers, editors, RDBMS, etc.). It is the developer's responsibility to acquire these items as needed.

Developers may install the toolkit on the disk in whatever directories are desired. The standard location for toolkit components is:

C public header files	`/h/COE/include`
Ada public package definitions	`/h/COE/include`
public libraries	`/h/COE/lib`
executables	`/h/TOOLS/bin`
UNIX `man` pages	`/h/TOOLS/man`
HTML documentation	`/h/TOOLS/HTML`

Certain tools from Appendix C are useful for both the development environment and the runtime environment. These tools are delivered with the operational system and are located under `/h/COE/bin`.

Developers should include `/h/TOOLS/bin` in the `path` environment variable for their development environment. `/h/TOOLS/man` should also be included in the search path for UNIX man pages. The web browser should be set to find HTML documentation under `/h/TOOLS/HTML`.

Developers are encouraged to submit tools to the COE community for inclusion in the developer's toolkit. All tools submitted must be license and royalty free, and must include a man page for online

[62] Documentation is delivered in only one format. The goal is to use HTML for programmer documentation because this is suitable for both NT and UNIX platforms. However, some documentation is still in UNIX `man` page format.

documentation. Developers wishing to release source code for their contributed tools may do so and the source code for the tool will be organized under the `/h/TOOLS/src` directory.

This page is intentionally blank.

10. COE Online Services

The DII COE provides a comprehensive set of services to assist in

- creating segments,
- tracking and managing submitted segments,
- tracking system trouble reports,
- distributing technical information and documents,
- communicating project-related information,
- distributing COE products to segment developers, and
- distributing COE-based systems to operational sites.

These services are provided by an SDMS and a COE Information Server (CINFO). The SDMS is an online software repository for receiving submitted segments, and for distributing them electronically, and for synchronizing repositories at mirror sites. The CINFO is used to disseminate project-related information including schedules and documentation. With appropriate restrictions, SDMS and CINFO services are available to segment developers, program managers, site administrators, services and agencies, and program sponsors.

Several network technologies are used to implement COE online services.

World-Wide-Web (WWW)	Access to catalogs, segments, plans, documents, etc. is provided via a WWW server. It is the standard interface to both SDMS and CINFO. Users will require a Hypertext Markup Language (HTML) browser such as Mosaic, Netscape, or Microsoft's Internet Explorer to access the WWW server.
Internet News	An Internet news server is used to manage newsgroups about the COE and COE-based systems. Such groups include technical discussions related to COE architecture, available tools, and standards.
anonymous ftp	Anonymous ftp servers are used to provide rapid dissemination of segments to operational sites. Sites may receive segments in either a "push" or a "pull" mode.

electronic mail	Automatic notification of key events (segment in test, segment ready for distribution, etc.) trouble reports, and meeting notices is done via electronic mail.

This approach provides several benefits to COE-based systems:

- Facilitates software and data reuse (e.g., segment reuse)
- Identifies available segments through a segment catalog
- Provides online configuration management
- Automates several aspects of the integration process
- Provides electronic notification of segment status to management
- Improves communications between segment developers
- Provides a centralized electronic distribution facility
- Separates classified or sensitive information from information suitable for general dissemination

Appendix D provides more information on how to access the COE online services described in this chapter.

10.1 Security Features

COE online services are separated into a classified and an unclassified system. The systems, whether classified or unclassified, use a secure operating system, database, and network software. Auditing is enabled to record system access and to record other security-relevant operations. Additional security features are implemented to

- ensure software integrity,
- prevent interception or eavesdropping on data transmissions, and
- ensure separation of classified versus unclassified information, segments, and data.

The classified and unclassified components reside on physically distinct computer systems separated by an air gap. The unclassified system is available via Internet and is generally available to any interested party. The classified system is accessible only via SIPRNET, and only to authorized users.

Unauthorized access to the system is prevented through a layered approach. Firewalls are implemented as the first layer of protection. Secure routers provide IP address filtering and port access to limit access only to authorized platforms. Features are also implemented to restrict services that can be requested or granted to further protect the system from unauthorized access.

User authentication is based on a combination of a manual registration process, an authorized IP address, and password protection. Passwords are required to initially log onto the system, but are further required to log into the software repository and to access browser services.

Public key encryption is used to protect segments in the software repository. Encryption and compression are both used to protect data during transmission over the network to prevent unauthorized modifications.

Certain information, such as system problem reports or project status, is not necessarily classified. However, such information is still sensitive and needs to be controlled. Public and private views are implemented to provide this measure of protection.

Further discussion of security features is beyond the scope of this document.

10.2 Software Distribution Management System

SDMS is the DII software repository, and it is used to store and disseminate COE and COE-related products. SDMS is accessible only from SIPRNET. Segments, technical documentation, APIs, the COE developer's toolkit, and segment abstracts are also stored in the repository, but as appropriate, they are mirrored on the unclassified Internet set for access by the general community.

Segments can be sent electronically to the DISA OSF through the `submit` program. Segments may also be sent to the OSF via tape. Tape is necessary to accommodate large segments (such as database segments) or classified segments. Electronically transmitted segments are compressed to reduce transmission time, and encrypted to provide security. Online software at the OSF receives the segment and places it into a protected directory until it is tested for conformance and to ensure that it is an authorized segment. Only then is the segment actually checked into the SDMS. This process is described in more detail in Chapter 3.

Segments are retrieved from the SDMS in a similar way. As segments are approved for release, they are placed in a protected directory that is accessible via an anonymous ftp, or through a network browser.

Developers who desire SDMS access must request access from DISA through their appropriate government program sponsor. Those without SIPRNET access may request COE products, such as the developer's toolkit, on tape media.

Distribution of COE-based systems to operational sites also uses the SDMS. Site administrators must request access from DISA through their appropriate government channels.

10.3 COE Information Server

The COE information server is used to disseminate information to the at-large COE community. The information server provides the following types of information:

- general product information
- meeting minutes
- briefings
- segment descriptions
- user documentation
- programmatic documentation
- problem reports.

An unclassified WWW home page available via the Internet provides access only to non-sensitive general information from these categories. The classified WWW home page is available only on SIPRNET and includes a list of all available segments, segment version and patch information, information on upcoming system changes, and special installation instructions.

All information posted on the information server requires prior approval by the DISA Engineering Office. Information to be posted must be submitted to the engineering office by the appropriate service/agency representative.

10.4 Mirror Sites

Project managers for COE-based systems will often have their own SSA and procedures for configuration management, development, and project communication. Services and government agencies may wish to implement the COE online services at their own selected sites to more directly support their program. Such SSA sites are called *mirror sites.* A mirror site contains a copy of the SDMS that is updated on a periodic basis (e.g., daily, weekly).

Mirror sites have all of the same capabilities as the central DISA site, subject to three restrictions:

1. Mirror sites are not allowed to submit COE-component segments to a mirror site SDMS. This ensures centralized configuration management of the COE through the DII COE SSA.

2. Mission-area segments that are part of a COE-based system being developed in cooperation with DISA (e.g., GCCS, GCSS) may be provisionally submitted to a mirror site SDMS.

3. Segments with APIs for which a mirror site is responsible may be provisionally submitted to the mirror site SDMS.

Submission of COE-component segments or mission-application segments for DISA COE-based systems is considered provisional until formally accepted by the DII COE SSA. These restrictions are required in order to avoid configuration management problems.